Microtec
Research Inc.

MCC68K

Reference Manual

100133-010

TRADEMARKS

Apollo™, DOMAIN/OS™, and AEGIS™ are trademarks of Apollo Computer, Inc.

Flexible License Manager™ is a trademark of Highland Software, Inc.

High C® is a registered trademark of Metaware, Inc.

HP™ and HP-UX™ are trademarks of Hewlett-Packard Company.

IBM® is a registered trademark of International Business Machines, Inc.

ICE® is a registered trademark of Intel Corporation.

Microtec® and Paragon® are registered trademarks of Microtec Research, Inc.

MRI ASM™, MRI C™, MRI FORTRAN™, MRI Pascal™, and MRI XRAY™ are trademarks of Microtec Research, Inc.

MS-DOS™ is a trademark of Microsoft Corporation.

pSOS™ is a trademark of Software Components Group.

Sun™ is a trademark of Sun Microsystems, Inc.

Tektronix® is a registered trademark of Tektronix, Inc.

UNIX® is a registered trademark of AT&T.

VAX™, VMS™, and ULTRIX™ are trademarks of Digital Equipment Corporation.

VRTX™ is a trademark of Ready Systems, Inc.

XRAY68K™, XRAY86™, XRAY180™, XRAYZ80™, XRAY29K™, XRAY51™, XRAY88K™, XRAY960™, XRAYG32™, XRAYTX™, XRAY Debugger™, XRAY In-Circuit Debugger™, and XRAY In-Circuit Debugger Monitor™, XHM302™, XRAYH83™, and XRAYH85™ are trademarks of Microtec Research, Inc.

Zilog® is a registered trademark of Zilog Corporation.

302bug™ is a trademark of Motorola, Inc.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 52.227-7013 of DOD Regulation 27-401, or comparable clause, as such may be amended or renumbered from time to time.

MICROTEC RESEARCH, INCORPORATED
2350 MISSION COLLEGE BLVD.
SANTA CLARA, CA 95054

© Copyright 1987, 1988, 1989, 1990, 1991 Microtec Research, Inc. All rights reserved. No part of this publication may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical or otherwise, without prior written permission of Microtec Research, Inc.

REV.	REVISION HISTORY	DATE	APPD.
-001	Original issue.	8/88	T.P.
-002	Corrections.	11/88	T.P.
-003	Revise to meet ANSI C standards.	4/89	T.P
-004	(Non-ANSI) Produced at 80% for new packaging.	12/89	T.P.
-005	ANSI MCC68K.	12/89	T.P.
-006	Added instructions for the VAX/VMS, Apollo DOMAIN/IX, and Motorola Delta Systems UNIX/System V systems to the documentation set.	2/90	T.P.
-007	Added support for VAX/ULTRIX and HP-UX.	4/90	J.A.
-008	Manual revision to support software Version 4.1.	6/90	T.P.
-009	Manual production upgraded from Framemaker version 1.3 to 2.0.	9/90	T.P.
-010	Manual revised to support version 4.2 software.	9/91	L.W.

Preface

About this Manual

This manual describes how to use the Microtec Research C compiler and related programs for the 68000 family of microprocessors.

The *Reference Manual* consists of the following chapters and appendices:

- Chapter 1, *Introduction*, describes the components of the C compiler.
- Chapter 2, *C Language Elements*, describes the elements (the names, numbers, and characters) used to write a C program.
- Chapter 3, *C Program Structure*, describes the structure of a C language source program.
- Chapter 4, *The Preprocessor*, describes preprocessor directives and the compiler's predefined symbols.
- Chapter 5, *Library Functions*, describes the run-time library functions available on the compiler.
- Chapter 6, *Internal Data Representation*, describes the data types used by the compiler.
- Chapter 7, *C Calling Conventions*, describes the assembly language calling conventions for the compiler. This chapter helps you write assembly language routines to interface with C language functions.
- Chapter 8, *Embedded Environments*, provides important information for users of embedded systems. It includes information on embedded applications, HP 64000 support, how to use the linker, position-independent code, and reentrant code. It also provides information on reserving a register, and direct memory and I/O port addressing.
- Chapter 9, *Optimizations*, describes the various kinds of optimizations that the compiler performs.
- Chapter 10, *Compiler Output Files*, describes how the compiler produces output files.
- Appendix A, *ASCII Character Set*, lists the ASCII character set.

- Appendix B, *Syntax of the ANSI C Language*, describes the lexical rules and syntax of the ANSI C language.
- Appendix C, *Error Messages*, describes diagnostic messages produced by the compiler.

Related Publications

This documentation assumes that you are familiar with programming and with the C programming language.

For general information about C, refer to the following publications:

- *The C Programming Language*, 2nd ed., by Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, Inc., 1988.

  This book contains an introduction to C with examples.

- *C — A Reference Manual*, 3rd ed., by Samuel P. Harbison and Guy L. Steele, Jr., Prentice-Hall Inc., 1991.

  This book contains a complete discussion of both ANSI C and traditional "K & R" C. It includes all library functions.

For additional information about MCC68K, refer to the following publication:

- *Getting the Most Out of Your Microtec Research MCC68K ANSI C Compiler*, Microtec Research, Inc.

  This document gives you tips to help you get the most out of your MCC68K Compiler.

For information about other Microtec Research 68K toolkit components, refer to the following publications:

- *ASM68K User's Guide, Reference Manual*, and *Installation Guide*, Microtec Research, Inc.

  These manuals describe how to use the ASM68K Assembler, the LNK68K Linker, and the LIB68K Librarian.

- *XRAY68K User's Guide, Reference Manual*, and *Installation Guide*, Microtec Research, Inc.

  These manuals describe how to use the XRAY68K Debugger.

Notational Conventions

The *Microtec Research MCC68K Reference Manual* uses the notational conventions shown in Table P-1 (unless otherwise noted).

Table P-1. Notational Conventions

Symbol	Name	Usage
{ }	Curly Braces	Encloses a list from which you must choose an item.
[ ]	Square Brackets	Encloses items that are optional.
...	Ellipsis	Indicates that you may repeat the preceding item zero or more times.
|	Vertical Bar	Separates alternative items in a list.
	Punctuation	Punctuation such as commas (,) and colons (:) must be entered as shown.
	`Typewriter Font`	Represents user input in interactive examples.
	Italics	Represents a descriptive item that should be replaced with an actual item.

Questions and Suggestions

To help us respond to questions or suggestions regarding the MCC68K Compiler, we have provided two response forms in the back of this documentation.

The first form is a Reader's Response sheet, which is used to help us correct and improve our documentation. We always appreciate your comments, and by completing this form, you can participate directly in the revisions of our publications. The Reader's Response sheet is directed to Technical Publications.

The second form is a Software Performance Report that should be completed if you encounter any errors or problems with Microtec Research software. This report is directed to Technical Support.

Contents

Preface

About this Manual v
Related Publications vi
Notational Conventions vii
Questions and Suggestions vii

1 Introduction

Overview of the Microtec Research Compiler 1-1
Compiler Package Features 1-1

2 C Language Elements

Overview 2-1
Identifiers 2-1
Constants 2-1
Integer Constants 2-1
Floating-Point Constants 2-2
Character Constants 2-4
Escape Sequences 2-4
String Literals 2-4
Operators 2-5
Keywords 2-5
Comments 2-6

3 C Program Structure

Overview 3-1
Variables 3-1
Data Type Summary 3-2
Scalar Data Types 3-2
Float and Double Types 3-2
Pointers 3-3
Void 3-3
Storage Classes 3-4
Storage Class Keywords 3-4
Storage Classes 3-5
Auto 3-5
Extern 3-5
Public 3-5
Register 3-5
Static 3-6

Sharing Variables 3-6
Expressions 3-6
Constant Expressions 3-7
Primary Expressions 3-7
Unary Expressions 3-8
Binary Expressions, Conditional Expressions, and Assignments 3-10
Standard C Operators 3-10
Precedence and Order of Evaluation 3-13
Type Conversion 3-15
Mixed Operands 3-15
Type Casting 3-16
Declarations 3-17
Type Qualifiers 3-18
Const 3-18
Volatile 3-18
Initialization 3-18
Type Definitions 3-19
Statements 3-19
Expression Statement 3-19
Compound Statement 3-20
break Statement 3-20
continue Statement 3-20
do-while Statement 3-20
for Statement 3-21
goto Statement 3-22
Labeled Statement 3-22
if Statement 3-22
if-else Statement 3-22
return Statement 3-23
switch Statement 3-23
while Statement 3-24
Functions 3-25
Function Return Types 3-25
Function Prototypes 3-25
Function Prototype Arguments 3-25
Function Declaration 3-26
Function Declaration With Body 3-27
Functions Used Without Being Declared 3-28
Using Prototype Function Declarations 3-29
Include Library Function Files 3-30

4 The Preprocessor

Overview 4-1
Predefined Symbols 4-1

Preprocessor Directive Groups 4-4
Source File Include Directive 4-4
Line Number Control Directive 4-4
Macro Definition Directives 4-5
Conditional Compilation Directives 4-6
#pragma Directives 4-7
Preprocessor Directives 4-7
Syntax Categories 4-7
Error_message 4-7
Expression 4-8
File_name 4-8
Include_name 4-9
Line_number 4-9
Macro_name 4-9
Option_list 4-9
Parameter_namelist 4-9
Pragma_directive 4-9
Replacement_text 4-9
#define — Defines a Macro 4-11
#elif — Ends the True Part of a Conditional Construct and Begins Another Conditional Construct 4-12
#else — Ends a Sequence of Conditional Constructs and Starts the False Part 4-13
#endif — Terminates a Conditional Construct 4-14
#error — Issues an Error Message then Halts Compilation 4-15
#if — Begins a Conditional Construct 4-16
#ifdef — Tests Whether a Macro Name is Defined 4-17
#ifndef — Tests Whether an Identifier is Not Defined 4-18
#include — Declares an Include File 4-19
#line — Changes the Line Number 4-20
#pragma asm — Begins Embedded Assembly Instructions 4-21
#pragma endasm — Ends Embedded Assembly Instructions 4-21
#pragma list — Suppresses Source Line Listings 4-22
#pragma option — Specifies Command-Line Options 4-23
#undef — Cancels the Definition of a Macro Name 4-24

5 Library Functions

C Function Groups and Include Files 5-1
assert.h 5-3
ctype.h 5-3
errno.h 5-4
float.h 5-5
limits.h 5-5
locale.h 5-5

math.h 5-5
mathf.h 5-7
mriext.h 5-9
setjmp.h 5-11
signal.h 5-11
stdarg.h 5-12
stddef.h 5-12
stdio.h 5-13
stdlib.h 5-17
string.h 5-18
time.h 5-19
System Functions 5-20
Non-Reentrant Functions 5-21
Library Function Definitions 5-22
abort — Causes Abnormal Program Termination 5-23
abs — Computes Absolute Value of an Integer 5-24
acos — Computes Arc Cosine of a Number 5-25
asctime — Converts Date and Time to String 5-26
asin — Computes Arc Sine of a Number 5-27
assert — Tests Program Expressions 5-28
atan — Computes Arc Tangent of a Number 5-29
atan2 — Computes Arc Tangent d1/d2 (with Overflow Protection) 5-30
atexit — Queues Function to be Called at Program Termination 5-31
atof — Converts ASCII String to Floating-Point Number 5-32
atoi — Converts ASCII String to Integer 5-33
atol — Converts ASCII String to Long Integer 5-34
bsearch — Performs Binary Search 5-35
calloc — Allocates Data Space Dynamically 5-36
ceil — Computes Integer Ceiling of Double 5-37
clearerr — Clears the File I/O Error Indicator 5-38
clock — Returns Processor Time Used 5-39
close — Closes a Specified File 5-40
cos — Computes Cosine of a Number 5-41
cosh — Computes Hyperbolic Cosine of a Number 5-42
creat — Creates a Specified File 5-43
ctime — Converts Date and Time to a String 5-44
difftime — Computes Difference Between Two Calendar Times 5-45
div — Computes Quotient and Remainder for a Division 5-46
eprintf — Provides Formatted Print to Standard Error 5-47
exit — Closes All Opened Files and Terminates Program 5-48
_exit — Terminates a Program 5-49
exp — Computes Exponential Value 5-50
fabs — Computes Absolute Value of a Floating-Point Number 5-51

fclose — Closes a File 5-52
feof — Checks for End-Of-File on a File 5-53
ferror — Checks the File I/O Error Indicator 5-54
fflush — Flushes Buffered Data to a File 5-55
fgetc — Reads a Character from a Specified File 5-56
fgetpos — Gets the Current Location in a File 5-57
fgets — Reads a String from a Specified File 5-58
fileno — Gets File Descriptor 5-59
floor — Rounds a Floating-Point Number to Nearest Integer 5-60
fmod — Computes Floating-Point Remainder 5-61
fopen — Opens a File 5-62
fprintf — Provides Formatted Print to a Specified File 5-64
fputc — Writes a Character to a Specified File 5-65
fputs — Writes a String to a Specified File 5-66
fread — Reads from a Specified File 5-67
free — Frees Dynamically Allocated Data Space 5-68
frexp — Breaks Down Floating-Point Number into Mantissa and Exponent 5-69
fscanf — Reads from a Specified File 5-70
fseek — Sets the Current Location in a File 5-71
fsetpos — Sets the Current Location in a File 5-72
ftell — Gets the Current Location in a File 5-73
ftoa — Converts Floating-Point Number to ASCII String 5-74
fwrite — Writes to a Specified File 5-75
getc — Reads a Character from a File 5-76
getchar — Reads a Character from Standard Input 5-77
getl — Reads a Long Integer from a Stream 5-78
gets — Reads a Line from Standard Input 5-79
getw — Reads a Short Integer from a Stream 5-80
gmtime — Breaks Down Calendar Time to Local Time 5-81
isalnum — Tests for an Alphanumeric Character 5-82
isalpha — Tests for an Upper- or Lower-Case Character 5-83
isascii — Tests for an ASCII Character 5-84
iscntrl — Tests for a Control Character 5-85
isdigit — Tests for a Digit 5-86
isgraph — Tests for a Printable Argument 5-87
islower — Tests for a Lower-Case Character 5-88
isprint — Tests for a Printable Argument (Includes Blank Character) 5-89
ispunct — Tests for a Punctuation Mark 5-90
isspace — Tests for White Space 5-91
isupper — Tests for Upper-Case Character 5-92
isxdigit — Tests for a Hexadecimal Digit 5-93
itoa — Converts Integer to ASCII String 5-94
itostr — Converts Unsigned Integer to ASCII String 5-95

labs — Computes Absolute Value of a Long Integer 5-96
ldexp — Computes Product of a Floating-Point Number and a Power of Two 5-97
ldiv — Computes Quotient and Remainder for a Division 5-98
localtime — Breaks Down Calendar Time to Local Time 5-99
log — Computes Natural Logarithm of a Number 5-100
log10 — Computes Base 10 Logarithm of a Number 5-101
longjmp — Jumps Outside of a Function 5-102
lseek — Sets the Current Location in a File 5-103
ltoa — Converts Long Integer to ASCII String 5-104
ltostr — Converts Unsigned Long Integer to ASCII String 5-105
malloc — Allocates Data Space Dynamically 5-106
max — Returns the Greater of Two Values 5-107
memccpy — Copies Characters from Memory 5-108
memchr — Searches Character in Memory 5-109
memclr — Clears Memory Bytes 5-110
memcmp — Compares Characters in Memory 5-111
memcpy — Copies Characters from Memory 5-112
memmove — Copies Characters from Memory 5-113
memset — Sets the Value of Characters in Memory 5-114
min — Returns the Lesser of Two Values 5-115
modf — Breaks Down a Number into Fractional and Integer Parts 5-116
offsetof — Determines Offset of Structure Member 5-117
open — Opens a Specified File 5-118
perror — Writes Error Message to Standard Error Device 5-120
pow — Computes d1 Raised to the Power d2 5-121
printf — Formatted Print on Standard Output 5-122
putc — Writes a Character to a File 5-126
putchar — Writes a Character to Standard Output 5-127
putl — Writes a Long Integer to a Stream 5-128
puts — Writes a String to Standard Output 5-129
putw — Writes a Short Integer to a Stream 5-130
qsort — Performs Quick Sort Algorithm 5-131
rand — Pseudo-Random Number Generator 5-132
read — Reads Bytes from a Specified File 5-133
realloc — Changes the Size of Dynamically Allocated Storage 5-134
remove — Makes File Name Inaccessible 5-135
rewind — Points to Beginning of File 5-136
sbrk — Allocates Memory Space 5-137
scanf — Provides Formatted Read from Standard Input 5-138
setbuf — Associates a Buffer with an I/O File 5-141
setjmp — Establishes a Label for longjmp 5-142
setlocale — Sets or Queries Local Information 5-143
sin — Computes Sine of a Number 5-144

sinh — Computes Hyperbolic Sine of a Number 5-145
sprintf — Formats a String 5-146
sqrt — Computes Square Root of a Number 5-147
srand — Sets the Seed of the Random Number Generator 5-148
sscanf — Provides Formatted String Conversion 5-149
strcat — Concatenates Two Strings 5-150
strchr — Locates First Occurrence of a Character in a String 5-151
strcmp — Compares Two Strings 5-152
strcoll — Compares Two Strings 5-153
strcpy — Copies a String to Another String 5-154
strcspn — Computes Initial Segment Length of String 5-155
strerror — Points to Error Message 5-156
strlen — Returns String Length 5-157
strncat — Performs Limited Concatenation of Two Strings 5-158
strncmp — Performs Limited Comparison of Two Strings 5-159
strncpy — Performs Limited Copy of a String 5-160
strpbrk — Finds Break Character in String 5-161
strrchr — Locates Last Occurrence of a Character in a String 5-162
strspn — Computes Initial Segment Length of String 5-163
strstr — Finds First Occurrence in String 5-164
strtod — Converts ASCII String to Double 5-165
strtok — Locates Tokens in a String 5-166
strtol — Converts ASCII String to a Long Integer 5-167
strtoul — Converts String to Unsigned Long Integer 5-168
swab — Swaps Odd and Even Bytes in Memory 5-169
tan — Computes Tangent of a Number 5-170
tanh — Computes Hyperbolic Tangent of a Number 5-171
time — Determines Current Calendar Time 5-172
toascii — Converts a Byte to ASCII Format 5-173
tolower — Converts Characters to Lowercase 5-174
_tolower — Converts Characters to Lowercase Without Argument Checking 5-175
toupper — Converts Characters to Uppercase 5-176
_toupper — Converts Characters to Uppercase Without Argument Checking 5-177
ungetc — Returns a Character to a File 5-178
unlink — Unlinks a File Name 5-179
va_arg — Returns Arguments from a Variable Argument List 5-180
va_end — Terminates Access to a Variable Argument List 5-182
va_start — Allows Access to a Variable Argument List 5-183
vfprintf — Provides Formatted Print to a Specified File 5-184
vprintf — Provides Formatted Print on Standard Output 5-185
vsprintf — Formats a String 5-186
write — Writes Bytes to a Specified File 5-187

zalloc — Allocates Data Space Dynamically 5-188

6 Internal Data Representation

Overview 6-1
Storage Layout 6-1
Data Type Summary 6-3
Scalar Data Types 6-4
char 6-4
unsigned char 6-4
short int 6-5
unsigned short int 6-6
int 6-6
unsigned int 6-7
long int 6-8
unsigned long int 6-8
Data Types — Float and Double 6-9
Float 6-10
Double and Long Double 6-12
Pointers 6-13
Bit Fields 6-14
Enumeration Types 6-16
Arrays 6-17
Strings 6-18
Structures 6-18
Unions 6-20
Alignment and Packing 6-21
Packing Bit Fields 6-21
Alignment of Bit Fields 6-21
Aggregates 6-22
Size of Aggregates 6-22
Packed Structures 6-23
Packed Enumerator Types 6-24
Allocation of Variables 6-25
Local Variables 6-25
Static Variables 6-25
Memory Allocation 6-26

7 C Calling Conventions

Overview 7-1
Parameter Passing 7-1
Setting Parameters 7-1
Setting Short Integers 7-1
Implicit Parameter for Structure/Union Return Value 7-2
Alignment of Structure/Union in Parameter Area 7-2

Function Return Values 7-3
Integer Return Values 7-3
Floating-Point Return Values (with FPU) 7-3
Floating-Point Return Values (without FPU) 7-3
Structure/Union Return Value 7-3
Popping Parameters 7-4
Implicit First Parameter for Structure/Union Return Value 7-4
Register and Stack Usage with Functions 7-4
Stack Frames 7-5
The Prologue 7-6
Local Variables in the Prologue 7-6
The Epilogue 7-7
Assembly Language Interfacing 7-7
Assembly Language Routine Example 7-8
Variable References from Assembly Routines 7-10
Interrupt Handlers 7-10

8 Embedded Environments

Embedded Applications 8-1
In-Line Assembly 8-1
Syntax and Constraints 8-1
Simple Use of asm 8-2
Returning a Typed Value 8-3
Using volatile with asm 8-3
Using #define for Readability 8-4
Variable Names Inside asm 8-4
Back Quoting Variable Names 8-4
Inserts and Global Variables 8-5
asm Versus #pragma asm/#pragma endasm 8-6
Other Examples 8-6
Guidelines for Assembler Inlining 8-9
Considerations for Assembler Inlining 8-9
Considerations for Embedded Systems 8-10
Removing Unneeded Floating-Point Support 8-10
System Functions 8-10
HP 64000 Systems 8-11
Using HP 64000 with the Microtec Research Compiler 8-11
Using HP 64000 with ASM68K 8-12
HP 64000 Requirements 8-13
User-Modified Routines 8-14
User-Modified Routines for Embedded Systems 8-14
User-Modified Routines for HP Systems 8-17
Using the Linker 8-18
Default Sections 8-18

Libraries 8-18
INITDATA Command (IEEE Only) 8-19
Linker Command Example (IEEE Only) 8-20
Linker Command Example (ROM-Based System, IEEE Only) 8-22
Linker Command Example (HP 64000 System, IEEE Only) 8-24
Position-Independent Code 8-27
Position-Independent Versus Position-Dependent Code 8-27
Compiler Considerations 8-28
Programmer Considerations 8-29
Assembler and Linker Considerations 8-32
Absolute Versus Register-Relative Addressing 8-32
Absolute Addressing 8-33
PC-Relative Addressing 8-35
Reserving a Register 8-37
Shared Program 8-38
Shared Data 8-39
System Data 8-40
Special Purposes 8-41
Reentrant Code 8-42
A Simple Solution 8-43
Multitasking/Multi-Threaded Environments 8-44
First Approach 8-44
Second Approach 8-48
Direct Memory and I/O Port Addressing 8-51
Calling Functions at an Absolute Address 8-51
Data Initialization 8-52
Saving Initialized Variables in ROM 8-52
Using the INITDATA Command (IEEE Only) 8-52
Code Organization 8-53
Changing Default Section Names 8-53
Changing Default Addressing 8-54

9 Optimizations

General Optimizations 9-1
Algebraic Simplification 9-1
Constant Folding 9-1
Strength Reduction 9-1
Redundant Code Elimination 9-2
Unreachable Code 9-2
Global Optimizations 9-3
Register Coloring 9-3
Factorization 9-3
Dead Code Elimination 9-3
Unused Definition Elimination 9-4

Global Constant Propagation 9-5
Global Copy Propagation 9-5
Branch Tail Merging 9-6
Local Optimizations 9-7
Common Subexpression Elimination 9-7
Redundant Load and Store Elimination 9-7
Generating Code for the Switch Statement 9-7
Local Constant Propagation 9-8
Local Copy Propagation 9-9
Loop Optimizations 9-9
Loop Invariant Code Optimization 9-10
Strength Reduction and Index Simplification 9-10
Loop Rotation 9-11
Array Operator Synthesis 9-12
Jump Optimizations 9-12
Redundant Jump Elimination 9-13
Multiple Jump Optimization 9-13
Code Hoisting 9-13
Cross-Jump Optimization 9-14
Short/Long Displacement Optimization 9-14
Function In-lining 9-15
Instruction Scheduling Optimization 9-15
Machine-Dependent Optimizations 9-16
Generating Code for Function Prologue and Epilogue 9-16
Grouping Stack Adjust Instructions 9-16
Indexing Arrays 9-17

10 Compiler Output Files

Assembler Source File 10-1
Advantages of Producing an Assembly File 10-1
Variable Names 10-1
External and Public Variable Names 10-1
Static Variable Names 10-2
Line Numbers 10-3
Code and Data Sections 10-3
Compiler Output Listing 10-3

Appendix A: ASCII Character Set A-1

Appendix B: Syntax of the ANSI C Language B-1

Appendix C: Error Messages

Overview C-1
Message Severity Levels C-1
Compiler Messages C-2

Figures

Figure 6-1. Word Ordering 6-1
Figure 6-2. Long Word Ordering 6-1
Figure 6-3. Byte Ordering in Words 6-2
Figure 6-4. Byte Ordering in Long Words 6-2
Figure 6-5. Position of the Most and Least Significant Bits 6-2
Figure 6-6. Sample Bit Field Allocation (over 4 bytes) 6-15
Figure 6-7. Sample Bit Field Allocation (over 2 bytes) 6-15
Figure 6-8. Packing Bit Fields 6-21
Figure 6-9. Signed Bit Fields 6-22
Figure 6-10. Packing of Aggregates 6-23
Figure 6-11. Unpacked and Packed Structures 6-24
Figure 7-1. Parameter Area of the Stack 7-1
Figure 7-2. Parameter Area and Short Integers 7-2
Figure 7-3. Parameter and Return Value Areas 7-2
Figure 7-4. Parameter Area and Structure/Union 7-3
Figure 8-1. Memory Configuration 8-21
Figure 8-2. Memory Configuration (ROM Based) 8-23
Figure 8-3. Memory Configuration (HP 64000 Based) 8-26
Figure 8-4. Position-Independent Code (Example 1) 8-30
Figure 8-5. Position-Independent Code (Example 2) 8-32
Figure 8-6. Absolute Addressing for Code and Data 8-33
Figure 8-7. Absolute Addressing Example 8-34
Figure 8-8. PC-Relative Addressing 8-35
Figure 8-9. PC-Relative Addressing Example 8-36
Figure 8-10. Shared Programs 8-38
Figure 8-11. Shared Data 8-39
Figure 8-12. System Data 8-40
Figure 8-13. Reserved Register as a Pointer to Data 8-41
Figure 8-14. Reserved Register as a Storage Area 8-42
Figure 8-15. Memory Configuration for Multi-Threaded Environments (Approach 1) 8-47
Figure 8-16. Memory Configuration for Multi-Threaded Environments (Approach 2) 8-50

Tables

Table 2-1. Floating-Point Syntax 2-3
Table 2-2. Escape Sequences 2-4
Table 2-3. Keywords 2-5
Table 3-1. Data Types 3-2
Table 3-2. Complex Data Types 3-2
Table 3-3. Arithmetic, Logical, and Bitwise Operators 3-11
Table 3-4. Relational Operators 3-12
Table 3-5. Assignment Operators 3-12
Table 3-6. Structure, Union, Pointer, and Array Operators 3-13
Table 3-7. Miscellaneous Operators 3-13
Table 3-8. Order of Operator Evaluation 3-14
Table 3-9. Function Declaration Differences 3-29
Table 4-1. Preprocessor Symbols Predefined by Options 4-4
Table 4-2. Preprocessor Directives 4-7
Table 4-3. Expression Operators (Highest Precedence First) 4-8
Table 5-1. assert.h Macros 5-3
Table 5-2. ctype.h Functions 5-4
Table 5-3. errno.h Macros 5-4
Table 5-4. locale.h Functions 5-5
Table 5-5. math.h Functions 5-6
Table 5-6. math.h Error Macros 5-7
Table 5-7. mathf.h Functions 5-8
Table 5-8. mathf.h Error Macros 5-9
Table 5-9. mriext.h Functions 5-10
Table 5-10. mriext.h Macros 5-11
Table 5-11. setjmp.h Functions 5-11
Table 5-12. setjmp.h Types 5-11
Table 5-13. stdarg.h Functions 5-12
Table 5-14. stdarg.h Types 5-12
Table 5-15. stddef.h Types 5-12
Table 5-16. stddef.h Macros 5-13
Table 5-17. stdio.h I/O Functions 5-14
Table 5-18. stdio.h Formatted I/O Functions 5-15
Table 5-19. Macros and Structures 5-16
Table 5-20. stdlib.h Functions 5-17
Table 5-21. string.h String Functions 5-18
Table 5-22. string.h Memory Functions 5-19
Table 5-23. time.h Functions 5-19
Table 5-24. System Functions 5-20

Table 5-25. Other Non-Reentrant Functions 5-21
Table 5-26. Conversion Specification Flags 5-123
Table 5-27. Conversion Characters for printf 5-124
Table 5-28. Conversion Characters for scanf 5-139
Table 6-1. Data Types 6-3
Table 6-2. Complex Types 6-3
Table 6-3. Representation of char 6-4
Table 6-4. Representation of unsigned char 6-5
Table 6-5. Representation of short int 6-5
Table 6-6. Representation of unsigned short int 6-6
Table 6-7. Representation of int 6-7
Table 6-8. Representation of unsigned int 6-7
Table 6-9. Representation of long int 6-8
Table 6-10. Representation of unsigned long int 6-8
Table 6-11. Representation of Single-Precision float 6-10
Table 6-12. Representation of double or long double 6-12
Table 7-1. Instructions for Removing Bytes from Stack 7-5
Table 8-1. HP Output File Names 8-11
Table 8-2. Section Names for HP 64000 Development Systems 8-12
Table 8-3. Microtec Research and HP Section Directives 8-13
Table 8-4. Modifications For Syntax Incompatibilities 8-14
Table 8-5. Section Names 8-18
Table 8-6. Position-Independent Code Options 8-27
Table 8-7. Absolute Versus Register-Relative Addressing 8-33
Table 8-8. Commands for Absolute Addressing Example 8-34
Table 8-9. Commands for PC-Relative Addressing Example 8-36
Table 8-10. Code and Data Allocation 8-53
Table A-1. ASCII Character Set A-1
Table C-1. Message Severity Levels C-1

Introduction 1

Overview of the Microtec Research Compiler

The Microtec Research C compiler for the Motorola 68000 family† of microprocessors is designed to assist in the development of efficient, portable, and easy-to-maintain C programs for microprocessor-based systems.

The compiler package is a full implementation of ANSI C.

The compiler generates assembly language code that can be further processed by the Microtec Research 68000 assembler.

Compiler Package Features

The Microtec Research C compiler package provides the following features:

- Implements ANSI C and the traditional C programming language, including the full preprocessor
- Compiles large C source modules
- Has options to specify search paths for standard and non-standard #include files
- Generates optimized code for all members of the 68000 family
- Generates position-independent code and data references
- Generates reentrant code
- Produces highly optimized object code
- Generates in-line 68881 floating-point instructions or calls to library routines
- Has a simple interface to assembly language
- Supports in-line assembly instructions including referencing program variables by their C names
- Generates a listing file containing C code intermixed with assembly code

†. The 68000 family includes the 68000, 68008, 68010, 68012, 68020, 68030, 68040, CPU32, and 68881 microprocessors.

- Generates line number and symbolic debugging information for use with the Microtec Research XRAY Debugger
- Issues useful warning messages

C Language Elements 2

Overview

This chapter describes the elements of the C programming language. The C language elements contain the names, numbers, and characters used to write a C program. They include identifiers, constants, operators, keywords, and comments.

Identifiers

Identifiers are names given to variables, functions, and labels in a C language program.

Identifiers can contain letters, digits, and underscores, and must begin with either a letter or an underscore. The compiler distinguishes between upper- and lower-case characters in an identifier. Identifiers can be given any number of characters. The first 127 characters of external and global names must be unique. The compiler will prepend an underscore to all public symbols.

For more information, see the section *Modify Naming Conventions* in the *MCC68K User's Guide*.

Constants

A constant is an invariant quantity. Constants can be integers, floating-point values, character constants, enumeration identifiers, or string literals.

Integer Constants

A decimal integer constant is a sequence of numeric characters optionally preceded by a plus or a minus sign. Constants are assigned a type based on their value, as shown below.

$-(2^{31})$ to $(2^{31})-1$	int or long int
0 to $(2^{32})-1$	unsigned int or unsigned long int

Hexadecimal (base 16) integer constants begin with 0x or 0X and can contain the upper-case letters A through F or the lower-case letters a through f, in addition to the digits 0 through 9.

Octal (base 8) integer constants begin with 0.

Hexadecimal and octal constants are assigned a type based on their value, as shown below:

0x80000000 to 0x7fffffff	int or long int
0x00000000 to 0xffffffff	unsigned int or unsigned long int

Any integer constant can be followed by L or l, indicating that its type is long. Similarly, the suffix U or u indicates unsigned type, and the suffix UL or ul indicates unsigned long int.

Examples:

`0xEFF1`	Leading zero distinguishes a hexadecimal number from a symbol.
`123`	Decimal number of type integer.
`1231`	Decimal number of type long (ends in the letter "l", not the number "one").
`0xffffffff`	Hexadecimal number of type unsigned long int.
`-1UL`	The same value as 0xffffffff.
`0x100000000`	Illegal, out of range.
`0177777`	Octal number of type unsigned int, equal to 0xFFFF or 65535.
`+56u`	Decimal number of type unsigned int.

Floating-Point Constants

Floating-point constants have the following format:

[*sign*] *integer_part.fractional_part exponent*

Either the *integer_part* or the *fractional_part* (not both) can be omitted. Either the '.' or the *exponent* (not both) can be omitted. The syntax is described further in Table 2-1.

Table 2-1. Floating-Point Syntax

Object	Description
sign	An optional plus (+) or minus (-) sign.
integer_part	One or more decimal digits.
.	A decimal point.
fractional_part	One or more decimal digits.
exponent	The letter E (or e) followed by an optionally signed integer.

The value of a floating-point constant is the value of the *sign*, the *integer_part*, and the *fractional_part*, multiplied by 10 to the power given in the *exponent*. Float constants have a precision of 7 decimal digits. The values can range from $1.18 * 10^{-38}$ to $3.4 * 10^{+38}$ (decimal). Double constants have a precision of 15 decimal digits. The values can range from $9.46 * 10^{-308}$ to $1.79 * 10^{+308}$ (decimal).

Floating-point constants can have f or F as a suffix. Long double-precision floating-point constants can have l or L as a suffix.

Examples:

Constant	Equivalent Value
`+76.3e-14`	`.000000000000763`
`76.3E0`	`76.3`
`76.3E2`	`7630.`
`76.3`	`76.3`
`76.e5`	`7600000.`
`76.`	`76.`
`-0.3e1`	`-3.`
`0.3`	`0.3`
`0.`	`0.`
`.3`	`.3`
`1e5`	`100000.`
`-1.79e308`	Lowest value
`+1.79e308`	Highest value
`9.46e-308`	Smallest positive value

Character Constants

A character constant is a single ASCII character enclosed in single quotation marks. The value of a character constant is the one-byte ASCII value of the character.

Escape Sequences

When character constants are used, certain non-printable characters can be represented using escape sequences. Escape sequences are preceded by a backslash (\). The legal escape sequences are listed in Table 2-2.

Table 2-2. Escape Sequences

Sequence	ASCII Name	Hex Value	Description
\a	BEL	07	Bell (alert)
\b	BS	08	Back space
\f	FF	0C	Form feed
\n	NL	0A	Newline
\r	CR	0D	Carriage return
\t	HT	09	Horizontal tab
\v	VT	0B	Vertical tab
\'	'	27	Apostrophe or single quote
\"	"	22	Double quote
\\	\	5C	Backslash

To represent the numerical value of the character constant, up to three octal digits can follow the backslash in the escape sequence. Similarly, in hexadecimal, up to three hex digits can follow the backslash and the letter x in the escape sequence.

The compiler issues a diagnostic message if any other character follows a backslash.

String Literals

A string literal is a sequence of characters enclosed in double quotation marks; it is treated as an array of characters.

The escape sequences shown previously in Table 2-2 can also be used in string literals. For example, to have literal double quotation marks in a string literal, enclose the string in double quotation marks and use the backslash escape character as shown below.

```
"This is a \"string\" using embedded double quotation marks"
```

Note

Strings enclosed in double quotation marks are automatically terminated with a null byte. For example:

```
sizeof ("this")
```

yields a value of 5.

Operators

The Microtec Research C compiler supports the standard C language operators. These operators include arithmetic operators, relational operators, assignment operators, and structure, union, and array operators. Precedence of operators and their order of evaluation is described in the next chapter.

Keywords

Keywords are predefined identifiers that have special meaning to the C compiler. Other identifiers cannot conflict with these keywords. The keywords used by the C compiler are listed in Table 2-3.

Table 2-3. Keywords

Keywords				
auto	do	goto	return	typedef
break	double	if	short	union
case	else	int	signed	unpacked
char	enum	interrupt	sizeof	unsigned
const	extern	long	static	void
continue	float	packed	struct	volatile
default	for	register	switch	while

Note

The keywords **interrupt**, **packed**, and **unpacked** have been added by Microtec Research to implement extensions to the C compiler. Other identifiers cannot conflict with them. See *Enable Microtec Research Extensions* in the *User's Guide* for the command line option that can disable these keywords.

The keywords **const**, **signed**, and **volatile** were introduced by the ANSI C standard and are not present in older versions of the language. See *Set ANSI C Code Generation* in the *User's Guide* for the command line option that can disable these keywords.

Comments

Comments are used to document the code used in a program. A comment is a sequence of characters beginning with /* and ending with */. All characters that fall between these two symbols, including newlines, are ignored by the compiler.

Comments cannot be nested. An example of a comment is shown below.

```
/* This is a comment. */
```

C Program Structure 3

Overview

This chapter describes the structure of a C language source program. C programs contain variables, data types, storage classes, operators, expressions, declarations, statements, and functions. At the highest level, a C program is a sequence of variable and function declarations. Declarations specify the type and storage class. Variable declarations can have initial values; function declarations can contain a body of declarations and statements. Expressions appear in initial values and statements; operators appear in expressions.

A C program can be spread across several separately compiled source files, each containing completed declarations.

Variables

A variable is an object containing data that can be examined and modified by the program. A variable declaration assigns the variable a name, a data type, and possibly an initial value. In addition, all variables have an associated storage class that defines their visibility and lifetime.

Variables must be declared before they are used. The visibility (or scope) of a variable can be either local or global. Local variables are declared within a function and are not accessible to any other function. Global variables are declared outside of a function and are accessible throughout the rest of the source file.

The following sections describe the variable data types and storage classes.

Data Type Summary

The basic data types recognized by the Microtec Research C compiler are listed in Tables 3-1 and 3-2. For more information on data types, see Chapter 6.

Table 3-1. Data Types

Data Types	
char	unsigned char
double	long double
enum	
float	
int	
long	unsigned long
pointer	
short	unsigned short
unsigned	

Table 3-2. Complex Data Types

Complex Data Types	
array	struct
union	

Scalar Data Types

The scalar data types in the C language are **char**, **int**, **float**, and **double**. In the Microtec Research C compiler, **int** types have three sizes: **short** (16-bit), **int** (32-bit), and **long** (32-bit). An **int** type can be **signed** or **unsigned**. An **int** is **signed** if the keyword **unsigned** does not appear. If an **int** is declared with the keywords **short**, **long**, or **unsigned**, the keyword **int** can be omitted. The compiler represents all **int** data in two's complement form. With multi-byte storage, the most significant byte is stored at the lowest memory address.

Float and Double Types

The Microtec Research C compiler stores floating-point data types using the IEEE single- and double-precision format. The formats contain a sign bit field, an exponent field, and a fraction field.

The **long double** data type is treated like the **double** type.

Pointers

A pointer is a value corresponding to the address of an object in memory. A pointer may point to an object of any data type.

A pointer variable is declared by placing the pointer operator (*) in front of a variable name. The following are examples of variables declared as pointers.

Examples:

```
char *pc;    /* pc is a pointer to a character */
int  *ptr;   /* ptr is a pointer to an integer */
```

A pointer variable stores the address of an object it points to in memory. To obtain the address of an object, use the "address of" operator (&).

You can declare pointers to any type, including functions. In particular, if more than one pointer operator (*) precedes a variable name, the variable is declared as a pointer to a pointer.

Examples:

```
char **ppc;     /* ppc is a pointer to a pointer to a char    */
int  **ptr_ptr; /* ptr_ptr is a pointer to pointer to an int */
```

In Microtec Research C, the type pointed to must have been previously declared.

Void

The void type has no values or operations. It can specify the return type of a function which returns no values. For example, the `sort` function is a `void` type because it reorders a list of items but does not return any value.

Example:

```
/* sort v[0] ... v[n-1] into increasing order */
void sort(int v[], int n)
{
...
}
```

As a special case, `void` can also denote a function that has no parameters.

Example:

```
extern f(void);
```

Type `void` can also create generic pointers.

Example:

```
void *p;
int *i;
p = i;
```

The value of pointer `i` will be assigned to pointer `p`.

Storage Classes

When you declare an object (a variable or function), you also determine its visibility to the rest of the program and its lifetime. C combines the concepts of visibility and lifetime into the notion of "storage class". The storage class of an object is determined by the position of its declaration in the source file and by an optional storage class keyword.

A declaration is either "global" or "local". The term "global" refers to a declaration that appears outside of all function bodies. (In *The C Programming Language*, Kernighan and Ritchie use the term "external" rather than global, even though external is easily confused with the storage class **extern**, which is not the same.) A global declaration is visible throughout the rest of the source file, except where it is superseded by a local declaration with the same name.

Function declarations are always global; there are no nested functions.

The term "local" refers to a declaration that appears inside a function body. A local declaration is visible from the point of declaration to the end of the block.

Storage Class Keywords

The storage class keywords in C are **auto**, **extern**, **static**, **register**, and **typedef**. **typedef** is used like a storage class keyword, but it is not a storage class (it is described later in this chapter). There is also a storage class that has no keyword — the term "public" describes it.

Storage Classes

The storage classes of C are described below.

Auto

The auto storage class is used to minimize the amount of memory required by a program. Automatic (or auto) variables are always local. They are declared inside a function and are accessible only to that function. Storage for these variables is allocated on the stack when the function is called and released when the function returns. Local automatic variables can be declared with an initial value, which is assigned each time the function is entered.

If you do not specify a storage class in a local declaration, the auto storage class is assumed.

Extern

The extern storage class is used to allow access to a function that is declared public in another source file, or later in the same file. It is also used to refer to extern or public variables of the same name in this or other modules. Variables or functions can be declared extern inside or outside of a function.

Global variables with no storage class keyword and no initialization are equivalent to extern variables.

Public

The public storage class is our name for initialized objects with global declarations and no storage class keyword. It also refers to function declarations with no extern keyword. A public object is visible to all source files that declare it extern. It is an error to have two public objects with the same name in one program.

Register

The register storage class is used for local variables or parameters only. For each function, Microtec Research C allocates as many variables as possible to the hardware registers. A register variable can have an initial value, but the value is not retained between function calls. It is illegal to apply the "address of" operator to a register variable.

In general, the register keyword should be used to indicate which variables are heavily used and should reside in a register for better code efficiency.

Static

The static storage class can be used with global functions, or global or local variables. All static variables are allocated permanent storage. Thus, local static variables retain their values between function calls.

Global static variables and functions cannot be accessed by a function in another source file.

Local static variables can be declared with an initial value. Thus, they are initialized before the program starts. Functions which use local static variables might not be reentrant.

Sharing Variables

To guarantee that public variables are declared identically in all modules, the #include preprocessor directive can include their extern declarations in all modules. In addition, for each variable, there may be at most one module initializing it.

Example:

File: **globals.h**

```
extern type1 var1;
extern type2 var2;
...
extern typen varn;
```

File: **mod1.c**

```
...
#include "globals.h"
type1 var1 = 5;    /* public (initialized) in this */
...                /* module only */
```

File: **mod2.c**

```
...
#include "globals.h"
type1 var2 = 1;    /* public (initialized) in this */
...                /* module only */
```

Expressions

An expression is a sequence of one or more symbols, constants, and operators that evaluate to a value or an address. Any legal C expression is allowed by the Microtec Research C compiler.

Constant Expressions

Constant expressions can be calculated, in many cases, at compile time as well as at runtime. However, some statements (such as the initialization statement for static and global variables) require constant expressions to be calculated at compile time.

Primary Expressions

Primary expressions include identifiers, constants, array, or structure elements as shown below. Parenthesized expressions also fall into this category so that very complex expressions can be grouped as single operands. The syntax for primary expressions can be summarized as follows:

primary-expression:
identifier
constant
string_literal
(*expression*)

In addition, the following *postfix-expression* applies:

primary-expression [*expression*]
primary-expression (*expression* [, *expression*]...)
primary-expression.*identifier*
primary-expression-> *identifier*

For a complete description of the ANSI C syntax, see Appendix B.

The value of an identifier is the contents of the storage it represents. If the identifier is an array or function without square brackets ([ ]) or parentheses (( )), the value is the address of the zeroth element or the entry point of the function, respectively.

Constants and strings have the explicit values discussed earlier.

Brackets enclose an integral expression used to select an element of an array. The period operator (.) selects a member of a struct or union; the -> operator also selects a member of a structure, but its left operand is a pointer to a struct or a union.

A *primary-expression* followed by (*expression* [, *expression*]...) is a function call. The *primary-expression* is usually a function name, but it could also be a dereferenced pointer to a function. (Dereferencing is described later in this section.) The expressions (zero or more) passed to the function are evaluated and assigned to the formal parameters in the function declaration. Microtec Research C evaluates function arguments right to left; you should not write a function call whose

arguments affect the values of other arguments. A function returns a value which is then used in the expression in which it appears.

Examples:

```
i
a[i]
printf("the number is %d\n", a[i])
date.year
dateptr->year
matrix[i][j][k]
customer[n].credit_recptr->start_date.month
```

Unary Expressions

Unary expressions have the following form:

unary-expression:
'*' *expression*
'&' *lvalue*
'-' *expression*
'+' *expression*
'!' *expression*
'~' *expression*
'++' *lvalue*
'--' *lvalue*
lvalue '++'
lvalue '--'
'(' *type-name* ')' *expression*
sizeof *unary-expression*
sizeof '(' *type-name* ')'

For a complete description of the ANSI C syntax, see Appendix B.

Operator	Description
*	The * unary operator expects a pointer and returns the object to which it points. This is called dereferencing.
&	The & operator performs the reverse operation of the * operator, yielding the address of an *lvalue* rather than its contents.
-	The - symbol negates its operand.

+	The + symbol parallels the - operator. It evaluates to its operand's value.
~	The ~ symbol complements each bit.
!	The ! symbol yields zero (false) if its operand is non-zero (true); otherwise, it yields 1 (true).
++	The ++ operator increments its operands and assigns the result. If the operator appears before the operand, the expression has the incremented value. Otherwise, the expression has the original value of the operand.
--	The -- operator decrements its operands and assigns the result. If the operator appears before the operand, the expression has the decremented value. Otherwise, the expression is decremented after evaluation.
lvalue	An *lvalue* is an expression that represents a memory location and can be assigned a value.
type-name	A *(type-name)* is called a "type cast"; it causes its operand to be treated as a value of the given type rather than its own type.
sizeof	The sizeof operator returns the size of its operand in bytes.

Examples:

```
*pi                 /* The object to which pi points. */

&i                  /* The address of i.*/

++j                 /* Increment j and use the incremented value. */

j++                 /* Use the value of j and then increment j. */

*pc++               /* Yield the object to which pc points,*/
                    /* then increment pc (not *pc).*/

(*pc)++             /* Yield the object to which pc*/
                    /* points, then increment that object.*/

sizeof(ar)/sizeof(ar[0])
                    /* Yield the number of elements in */
                    /* the array ar.*/

(unsigned)i/2       /* Treat i as unsigned before dividing. */

(*apf[i])(x,y)      /* Call the function pointed to by apf[i], */
                    /* passing x and y as parameters. */
```

Binary Expressions, Conditional Expressions, and Assignments

Binary expressions, conditional expressions, and assignments have the following forms:

expression binary-op expression

expression ? *expression* : *expression*

lvalue assign-op expression

For a complete description of the ANSI C syntax, see Appendix B.

Binary operators are listed in Table 3-3.

The binary logical operators **&&** and **||** do not evaluate their second operands if the result of the operation is determined by the first operand. Specifically, if the first operand of **&&** is zero (false), the second operand is not evaluated and the value of the operation is zero (false). Similarly, if the first operand of **||** is non-zero (true), the second operand is not evaluated and the value of the operation is 1 (true). This "short-circuit" evaluation is efficient and can be used to prevent invalid address calculations, as in the following:

```
while ((i <= lim) && (a[i] != terminator)) {i++;}
if ((p == NULL) || (*p == terminator)) return;
```

Relational operators return 1 if the relation is true; they return 0 if false.

The assignment operator, **=**, stores the value of the right expression into the "lvalue" on the left. Assignment operations of the form **a op= b** are equivalent to **a = a op b**, but they only calculate the address of **a** once.

The conditional operator **?:** is the only three-operand (or ternary) operator. The operation **a?b:c** returns **b** if **a** is non-zero; otherwise, it returns **c**. The operation does not evaluate **c** if it returns **b**, and similarly does not evaluate **b** if it returns **c**.

The binary operation **a,b** evaluates **a**, then **b**, and returns **b**.

Standard C Operators

The standard C language operators are listed in the following tables:

- Table 3-3 lists the arithmetic, logical, and bitwise operators.
- Table 3-4 lists the relational operators.
- Table 3-5 lists the assignment operators.

- Table 3-6 lists the structure, union, pointer, and array operators.
- Table 3-7 lists the miscellaneous operators.
- Table 3-8 lists the order of evaluation of the operators.

Table 3-3. Arithmetic, Logical, and Bitwise Operators

Symbol	Description
+	Add
-	Subtract
*	Multiply
/	Divide
%	Modulo
+	Unary Add
-	Unary Negate
!	Logical NOT
&&	Logical AND
&	Bitwise AND
||	Logical OR
|	Bitwise OR
^	Bitwise Exclusive OR
~	Unary NOT
<<	Left Shift
>>	Right Shift
++	Pre-Increment/Post-Increment
--	Pre-Decrement/Post-Decrement

Table 3-4. Relational Operators

Symbol	Description
>	Greater Than
>=	Greater Than or Equal
<	Less Than
<=	Less Than or Equal
==	Equal
!=	Not Equal

Table 3-5. Assignment Operators

Symbol	Description
=	Assignment
+=	Add and Assign
-=	Subtract and Assign
*=	Multiply and Assign
/=	Divide and Assign
%=	Take Modulo and Assign
>>=	Shift Right and Assign
<<=	Shift Left and Assign
&=	Bitwise AND and Assign
^=	Bitwise Exclusive OR and Assign
|=	Bitwise OR and Assign

Table 3-6. Structure, Union, Pointer, and Array Operators

Symbol	Description
&	Address of
*	Pointer Dereference
->	Structure Pointer Reference
.	Structure Reference
[]	Array Reference

Table 3-7. Miscellaneous Operators

Symbol	Description
?:	Conditional Expression
()	Parenthesis for Grouping
,	Grouping
(*type*)	Casting
sizeof	Size Calculation in Bytes

Precedence and Order of Evaluation

The sequence of operations in an expression is determined by the C language's rules of precedence. If an operand appears between two operators, it is used by the operator with higher precedence. For example, a+b*c is equivalent to a+(b*c) because * has higher precedence than +. If an operand appears between two operators with the same precedence, it is grouped with the left or right operator, depending on the order of association for those operators.

Table 3-8 summarizes the rules of precedence and the order in which operators are evaluated. Operators are listed in decreasing precedence, and those on the same line have the same precedence.

Table 3-8. Order of Operator Evaluation

Operator	Order of Association
() [] -> .	left to right
! ~ ++ -- + - (*type*) * & sizeof	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	right to left
= += -= etc.	right to left
,	left to right

Unary operators have the highest precedence and group right to left. Binary operators have the next highest precedence and group left to right. The ternary operator has precedence immediately below the binary operators. The assignment operator has a precedence immediately below the ternary operator. The comma operator has the lowest priority and groups left to right.

You can override the rules of precedence and associativity by using parentheses. It is also useful to insert parentheses in complicated expressions to improve readability. Since the C language does not specify the order in which the operands of an operator are evaluated (except for &&, ||, ?:, and ,), code that depends on the order of evaluation should place intermediate values into temporary variables.

Type Conversion

The compiler performs data type conversions under the following circumstances:

- When arguments are passed to a function. In this case, the types of the function's arguments are converted as described in the next section, *Mixed Operands*.
- When the data type of an operand is forced to be converted by type casting.
- Many operators cause conversions and yield result types in a similar way. This pattern is called the "usual arithmetic conversions".
- When two or more operands of different types appear in an expression. The right operand is converted to the type of the left operand. The conversion of mixed operands is described below.

Mixed Operands

Mixed operands are converted as follows:

1. If either operand has type **long double**, the compiler converts the other to **long double**.
2. If either operand has type **double**, the compiler converts the other to **double**.
3. If either operand has type **float**, the compiler converts the other to **float**.
4. If either operand has type **unsigned long**, the compiler converts the other to **unsigned long**.
5. If one operand has type **long** and the other has type **unsigned**, the compiler converts both to **unsigned long**.
6. If either operand has type **long**, the compiler converts the other to **long**.
7. If either operand has type **unsigned**, the compiler converts the other to **unsigned**.

If the operands do not fit into the preceding type categories, the compiler performs the following "integral promotions" to ensure that both operands have type **int**.

1. Each **char** operand is sign-extended to **int**, unless the **/unsignedchar** (VMS) or **-Ku** (UNIX/DOS) option is specified.
2. Each **signed char** operand is sign-extended to **int**.
3. Each **unsigned char** operand is zero-extended to **int**.

4. Each **short** operand is sign-extended to **int**.

5. Each **unsigned short** operand is zero-extended to **int**.

Type Casting

Type casting forces the conversion of an expression to the specified data type. Casting to a type of the same size causes the same bit pattern to be interpreted as the new type. Casting to a larger type causes sign-extension or zero-padding according to the above rules.

When casting an integral type to a smaller integral type, the low-order bits are retained.

When casting a floating-point type to another floating-point type, the high-order bits of the fractional part are retained.

When casting an integral type to a floating-point type (or vice versa), the bits change completely.

Examples:

```
(char *) 5              /* Treat 5 as a pointer to a char. */

(unsigned) -1           /* Value: 4294967295 */

(int) 65535             /* Value: 65535 */

(long) -128             /* Value: -128, 4-byte size */

(unsigned long) -128    /* Value: 4294967168 */

(char) 4294967167       /* Value: 127 */

p2 = (type2 *) p1;      /* Assign p1's value to p2, */
                        /* even though p2 points to a */
                        /* different type. */
```

Declarations

Variables must be declared before they are used. The form of a variable declaration is:

declaration:

declaration-specifiers declaration-list;

declaration-specifiers:

storage-class-specifier
type-specifier
declaration-specifiers storage-class-specifier
declaration-specifiers type-specifier

declaration-list:

declaration
declaration-list declaration

For a complete description of the ANSI C syntax, see Appendix B.

Examples:

```
static int i, j;

extern char *pc, ch;

auto unsigned long *ular[sizeof(uli)*8];
                        /* Array of pointers to longs. */

int (*fptr)();          /* Pointer to a function returning */
                        /* an int. */

extern int *func();   /* Function returning pointer to int. */

static user_defined_type user_variable;

extern int *fpi;      /* Pointer to int. */

int const x = 3;

static const struct tag {int i; long l;} s;

long const volatile system_clock;
```

A declaration identifies a variable's name and type. In the example above, `i` is declared to be a `static int` variable, while `ch` is an `extern char` pointer variable. Similarly, `char *pc` means that `*pc` is a `char`; therefore, `pc` is a pointer to a `char`. Note that `()` and `[]` are applied before `*`, unless parentheses override. For example, in the declaration `char *ap[5]`, `ap` is an array, but in `char (*ap)[5]`, it is a pointer.

Type Qualifiers

There are two type qualifiers: const and volatile.

Const

You can prevent objects from having their values changed by declaring them to be of type const.

Volatile

Declaring a variable volatile makes it unavailable for optimization. For example, variables whose values can be altered by events not under the control of the program should be declared volatile.

Initialization

If not explicitly initialized in the definition, all static and extern variables have an initial value of zero. Entire arrays, structures, or groups of variables can be initialized in a single declaration. An initializer provides the initial values, which must be constant expressions unless the storage class is auto.

The syntax of an *initializer* in a declaration is as follows:

initializer:

expression
{ *initializer-list*, ... }

For a complete description of the ANSI C syntax, see Appendix B.

Examples:

```
auto aval = i*j;        /* Only auto allows */
                        /* non-constant initial values. */

int i = 5, j = 6, k = 7;
                        /* Three initializations. */

char hello[] = "hello, world!";
                        /* Store the characters in the array, */
                        /* giving the array a size of 14. */

char *hello = "hello, world!";
                        /* Store the characters in static */
                        /* storage, and initialize the */
                        /* pointer to point to them. */
```

```
itablerec itable[MAX_ITABLE+1] =
                /* An array of records. */
        {
        {0},
        {1, 22, 7, 0, 0, 0, 0, 0, 0, 0, 2, 3, 6, 1,
        0x0, 0x0, 0x0, 0x0, 0x2, 0x0, 0x2, 0x2, 0x2, 0x0, 0x1, 0x3 }
        };
```

Type Definitions

You can create your own names for types by using **typedef** as the storage class keyword. No storage is created, but you will be able to use the name you declare wherever a type name is allowed.

Example:

```
typedef struct { float rho, theta; } POLAR;
        /* Define POLAR. */

POLAR point1, endpoints[10];
        /* Declare POLAR-type variables. */
```

Statements

Statements evaluate expressions or control the flow of program execution. They also control loops, transfer flow control, and select other statements to be executed.

The keywords used in C statements are:

break	do	else	if	switch
continue	for	goto	return	while

Expression Statement

An expression statement has the following form:

expression;

When this statement is executed, *expression* is evaluated. Expression statements are only useful if they have side effects such as assignments or function calls.

Examples:

```
a = b + c;
++i;
printf("hello, world\n");
c=getchar();
b+c;          /* useless */
```

Compound Statement

Any sequence of declarations and statements can be enclosed by curly braces ({ }) and treated as a single statement. They are referred to as compound statements and can be used in the following control structures whenever a statement occurs.

break Statement

The **break** statement:

```
break;
```

terminates the execution of the smallest enclosing **do**, **for**, **switch**, or **while** statement in which it appears. Control passes to the statement following the terminated statement.

continue Statement

The **continue** statement:

```
continue;
```

can be used inside of a **do**, **for**, or **while** statement. It terminates the current loop iteration and causes the next iteration of the loop to execute.

do-while Statement

The **do-while** statement executes *statement* and then evaluates the expression after the keyword **while**. If *expression* is non-zero, *statement* is executed again and *expression* re-evaluated. This cycle continues until the *expression* evaluates to zero.

Curly braces only are required if there is more than one statement in the **do-while** loop.

```
do {
      statement;        /* execute this statement */
      [statement;]...   /* additional statements */
} while (expression);   /* while this expression is non-zero */
```

for Statement

The for statement has the following syntax:

```
for
      ([expression1];      /* evaluate only once */
      [expression2];       /* evaluate before each iteration */
      [expression3])       /* evaluate after each iteration */
{
      [statement;]...      /* execute statement(s) */
                           /* while expression2 is true */
}
```

The for statement is useful for executing a statement a specified number of times.

The for statement evaluates *expression1*, and then evaluates *expression2* to see if it is true (non-zero) or false (zero). If *expression2* is true, all *statements* are executed once. Then, *expression3* is evaluated, and *expression2* is evaluated again to see if it is true or false. If *expression2* is true, all *statements* are executed again and the cycle continues. If *expression2* is false, all *statements* in the `for` loop are bypassed and execution continues at the next statement outside the `for` loop.

In most cases, *expression1* initializes a variable to be used in the loop. *Expression2* compares the variable to some limit, and *expression3* increments or decrements a loop counter.

Curly braces only are required if there are multiple statements in the for loop.

Examples:

```
for (i=1; i<=10;++i) statement;
      /* execute statement with i= 1,2,...10 */

for (i=1, j=10; i<j; ++i,--j) statement;
      /* execute statement with */
      /* (i,j)=(1,10),(2,9),...(5,6) */

for (;;) statement;
      /* infinite loop, empty expression 2 is never */
      /* zero */
```

goto Statement

The goto statement:

```
goto label;    /* transfer control to label */
```

unconditionally transfers control to a labeled statement. The *label* (identifier) must be in the function containing the goto statement.

Labeled Statement

Any statement can be labeled by preceding it with a new identifier and a colon.

```
label: statement;          /* define as a label */
```

A label serves as the target of a goto statement.

if Statement

A simple if statement has the form:

```
if (expression)            /* If this expression is non-zero */
{
      statement;           /* execute this statement */
      [statement;]...      /* additional statements */
}
```

The word if is always followed by an expression enclosed in parentheses. If *expression* results in a value of zero (false), the statement following the expression is bypassed. If *expression* results in a value of non-zero (true), the statement following the expression is executed.

Curly braces only are required if there is more than one statement.

if-else Statement

The if-else statement contains an else clause which specifies an alternate statement to execute when the expression in the if statement evaluates to zero.

```
if (expression)            /* If expression is true */
{
      statement1;          /* execute statement1 */
      [statement;]...      /* additional statements */
}
else                       /* If expression is false */
{
      statement2;          /* execute statement2 */
      [statement;]...      /* additional statements */
}
```

If *expression* is true (non-zero), *statement1* and any following statements are executed, but *statement2* and any following statements are not executed. If *expression* is false (zero), *statement2* and any following statements are executed, but *statement1* and any following statements are not executed.

Curly braces only are required if there is more than one statement in the if or else section.

return Statement

The **return** statement:

```
return;
```

or

```
return (expression);
```

terminates the execution of the function in which it appears and returns control to the calling function. Execution resumes in the calling function at the point immediately following the call. The value of an expression, if present, is returned to the calling function. If an expression is not present, the return value of the function is undefined. Any statements in the function following the return statement are not executed.

switch Statement

The **switch** statement:

```
switch (expression) {
[case constant:
      [statement;]...     /* execute statement(s) */
      [break;]
]...
[default:
      [statement;]...     /* execute statement(s) */
]
}                         /* end of switch statement */
```

transfers control to one of several statements depending on the value of an expression. The expression is evaluated and the value is compared to the constant values following each `case` keyword. These values must be integer constants, character constants, or constant expressions.

Execution begins at the statement following the constant that matches the value of *expression*. If none of the constant values match the value of the expression, execution begins at the statement following the `default` keyword (if it appears). If

there is no default, none of the statements inside the `switch` statement are executed. Note that when a constant value is matched, all the following statements are executed. You must explicitly stop executing statements; using the **break** statement is the most common method.

Example:

```
switch (character1)
{
        case "a":
        case "A":       alpha = "a";
                        break;
        case "b":       alpha = "b";
                        break;
        case "B":       alpha = "B";
                        break;
        case "c":       alpha = "c";
                        break;
        ...
        case "z":       alpha = "z";
        default:        alpha = "0";
}
```

In the example above, `character1` is a user-defined variable of type **char**. If `character1` is the character "a", the compiler will go to the specified case entry and execute the statement `alpha = "a"`. The `break` statement causes the compiler to exit the `switch` statement. If `character1` is the character "z", the compiler will go to the specified entry and set the variable `alpha` equal to "z". Since there is no `break` statement, the compiler continues to the next statement (the `default` statement, in this case) and sets `alpha` equal to "0".

while Statement

The **while** statement:

```
while (expression)      /* while this expression is non-zero */
{
        [statement;]...     /* execute statement(s) */
}
```

evaluates an expression. If *expression* is non-zero, the statement is executed and the expression is re-evaluated. This cycle continues until the expression evaluates to zero.

Curly braces only are required if there is more than one statement.

Functions

All executable statements in a C program are contained in functions. A function is called from an expression (possibly with values passed as parameters). The function performs the computation and returns a result to the caller.

Functions can be declared, declared with a body, or used without being declared. In each case, the function call syntax is the same. The three forms of function declarations are described later in this section.

Function declarations can be in prototype form. Function return types and prototypes are described below.

Function Return Types

In ANSI C, functions can have return types, which can be specified by a data type preceding the function declaration. For example, a function declaration returning an integer value would be written like this:

```
int f(parameter_declarations)
```

Function Prototypes

A prototype specifies the number and type of arguments accepted by the function. Microtec Research strongly recommends using the prototype style for function declarations. For example, a function which returns the absolute value of an integer parameter, `i`, would be declared with the prototype:

```
int f(long i);
```

If a parameter is passed to `f` which is not compatible with type `long`, you will receive a diagnostic message.

Before you call a function, you can declare it explicitly with a prototype. When it is called, the function's arguments will be converted to the declared types of the formal parameter. The number and types of arguments must agree with the declared types. If they do not, you will receive a diagnostic message.

Function Prototype Arguments

Prototypes can specify no arguments, a fixed number of arguments, or a variable number of arguments.

A function that takes no arguments should use void as a parameter list. For example, if you have a function named `rand` which simply generates random numbers, you would declare:

```
extern int rand(void);
```

A function that takes a fixed number of arguments indicates the types of those arguments in a list. In the following example, the string comparison function, `strcmp`, expects two pointers to be of type `char`.

```
extern int strcmp (char *str, char *loc);
```

If the above definition appeared in a program, the following assignment statement would be legal:

```
j=strcmp("string","location");
```

The following statements, however, would be illegal:

```
j=strcmp();          /* illegal - the function contains no
                        arguments */
j=strcmp("string",1);/* illegal - the integer argument is
                        incompatible with the second
                        parameter, loc */
j=strcmp(3.0,2.5);   /* illegal - the arguments are float-
                        ing-point values */
j=strcmp("string","location","start");
                     /* illegal - the function has three ar-
                        guments */
```

Function Declaration

A function declaration contains the return type, the name of the function, and the optional parameter list. Using the example from the previous page, a function declaration which returns the absolute value of an integer parameter, `i`, would be declared with the prototype:

```
int abs(int i);
```

In function declarations, it is permissible to leave out the parameter name. Thus, it would be just as correct to write the following:

```
int abs(int);
```

Examples:

```
extern int rand(void);      /* no parameters */

long atol(char*);           /* with parameters */

long atol(char *cp)         /* with parameters */
{ statements }              /* body */

static float rand(void)     /* with parameters */
{ statements }              /* (can be empty) */

int (* functor())(int i)    /* functor returns pointer */
{ statements }              /* to an int function */

extern printf();            /* old style C */
                            /* Not a prototype, */
                            /* not recommended */
```

General Form:

function-declarator:

declarator (parameter-type-list,...)

declarator (parameter-list)

parameter-type-list:

parameter-declaration

parameter-type-list, parameter-declaration

parameter-declaration:

declaration-specifiers, declarator

declaration-specifiers, abstract-declarator

For a complete description of the ANSI C syntax, see Appendix B.

The differences between function declarations are summarized in Table 3-9.

Function Declaration With Body

Often, a function is declared with a body of code which defines it. Following the header, a function definition contains parameters and statements enclosed in curly braces ({ }). This is called the "body" of the function. For example,

```
f(int a);
{
   body
}
```

General Form:

function-definition:

declaration-specifiers declarator function-body

function-body:

parameter-declaration-list compound-statement

parameter-declaration-list:

declaration-list

declaration-list:

declaration

declaration-list declaration

For a complete description of the ANSI C syntax, see Appendix B.

The differences between function declarations are summarized in Table 3-9.

Functions Used Without Being Declared

When the program calls a function that has not been declared, the compiler will assume that a compatible prototype exists for it. For example, the `rand` function does not take any parameters and returns an integer type. It appears without a declaration in the program body below.

```
int f(int i)
{
   ...
   int r = rand( );
   ...
}
```

In the above example, the compiler will assume that the following compatible prototype exists for the `rand` function:

```
int rand(void);
```

Microtec Research does not recommend calling functions without prototypes. If you do, it is your responsibility to ensure that the prototype assumed by the compiler will be compatible with the function.

The differences between function declarations are summarized in Table 3-9.

Table 3-9. Function Declaration Differences

Function	Code

Function Declaration	`f(int i);`
	or
	`f(int);`

Function Definition with Body	`f(int a)` `{ body }`

Function Used Without Being Declared	`j= f(3);`
	`/* compiler assumes the */`
	`/* prototype int f(int i) */`
	`/* exists */`

Function Call to a Declared Function	`f(3.0);`
	`/* compiler converts 3.0 to int */`
	`/* since the prototype */`
	`/* specifies int */`

Using Prototype Function Declarations

One of the most significant features of ANSI C is the introduction of prototype function declarations. These declarations allow the C compiler to verify that functions are defined and used consistently.

Here is an example of a program that uses a prototype declaration for the library function `sqrt`:

```
extern double sqrt(double x);
      /* prototype declaration */
main()
{
      double r = sqrt(2);
}
```

The prototype declaration specifies that the parameter to `sqrt` has type `double`. When `sqrt` is called, the C compiler compares this type to the type of the argument `2`. Since the argument has int type, the compiler converts `2` to `double` format to ensure consistency with the declaration.

Before the invention of prototypes, the declaration of `sqrt` looked like this:

```
extern double sqrt();  /* old style declaration */
```

If you use old-style declarations, the compiler cannot ensure argument consistency. If this old-style declaration was used in the above program, the compiler would pass `2` as an int. Since `sqrt` assumes that its argument is in `double` format, it would yield an incorrect result.

Include Library Function Files

The sqrt function is provided by the ANSI C library (see Chapter 5). A prototype declaration for it can be found in the system include file **math.h**. Each function in the ANSI C library has its prototype declaration specified in one of the standard include files. To reduce the risk of a mistake, do not declare library functions yourself; include the appropriate file instead:

```
#include <math.h>
main()
{
   double r = sqrt(2);
}
```

Before ANSI C was developed, library functions such as **fprintf, free,** and **rand** were frequently used without being declared since their old-style declarations provided no additional information to the compiler. In ANSI C, their prototype declarations look like this:

```
extern int fprintf(FILE *stream, const char *format, ...);
                                 /* <stdio.h> */
extern void free(void *ptr);     /* <stdlib.h> */
extern int rand(void);           /* <stdlib.h> */
```

With ANSI C, it is always beneficial to include library declarations (from the appropriate include file). These declarations allow the compiler to issue diagnostic messages if they are called with too few arguments, too many arguments, or arguments of the wrong type. As illustrated above, the compiler always converts arithmetic arguments appropriately.

A function that takes a variable number of arguments (like `fprintf`) is declared with a `...` symbol at the end of its parameter list. This is necessary to stop the compiler's check for too many arguments.

A function that takes no arguments (like `rand`) is declared with a prototype of `void`. This is necessary to distinguish it from an old-style function declaration, which has empty parentheses.

Several library functions (like `free`) make use of the type `void*`. This type is compatible with all other pointer types; its use eliminates the casting that would otherwise be necessary when using generic pointers.

The Preprocessor 4

Overview

The Microtec Research C Compiler contains an ANSI C preprocessor. The Microtec Research C preprocessor is functionally identical to the UNIX C preprocessor. The preprocessor is not a separate pass, but is integrated into the compiler, thereby making the compilation process faster by requiring only one reading of the program.

Predefined Symbols

The preprocessor includes the following five predefined symbols:

__DATE__ — The value of __DATE__ is a character string that consists of the ASCII representation of the current date. It has the following format:

"*Mmm dd yyyy*"

Mmm has the same format as the month given by the asctime run-time function. *dd* has a leading blank if it is a one-digit number. For example, 5 May 1989 would be represented as "May 5 1989".

__FILE__ — The value of __FILE__ is a character string that consists of the ASCII representation of the current file name. The #line directive can be used to change the value of __FILE__.

__LINE__ — The value of __LINE__ is a decimal constant that indicates the current line number within the current file. The #line directive can be used to change the value of __LINE__.

__STDC__ — The value of the __STDC__ is the decimal constant 1. If __STDC__ is defined in a program, it designates that the program is written in ANSI C.

__TIME__ The value of __TIME__ is a character string that consists of the ASCII representation of the current time in 24 hour clock format. It has the following format:

"*hh:mm:ss*"

For example, 10:30:45 P.M. would be represented as "22:30:45".

The following symbols are also predefined by the preprocessor and can be used to customize target system-dependent code:

m68k	_SIZEOF_FLOAT
MC68000	_SIZEOF_INT
mc68000	_SIZEOF_LONG
_MCC68K	_SIZEOF_LONG_DOUBLE
_MRI	_SIZEOF_POINTER
_SIZEOF_CHAR	_SIZEOF_SHORT
_SIZEOF_DOUBLE	

Example:

```
#if _MCC68K
     printf("host compiler is MCC68K");
#else
     printf("host compiler is not MCC68K");
#endif
```

Example:

```
#if _SIZEOF_INT == 2
#error "this program is not suitable for 16-bit targets"
#endif
```

The following symbols are predefined by the preprocessor for compatibility with previous versions of Microtec Research C compilers and may be removed in the future:

MRI	mri
MCC68K	mcc68k

The following symbols describe the host environment and can be used to customize host compiler code:

_APOLLO	_M68	_SUN3
_BCS	_MSDOS	_SUN4
_DEC_STATION	_PC	_UNIX
_HP9000_300	_SCO	_VAX
_HP9000_700	_sun	_VMS
_HP9000_800		

_sun is a preprocessor symbol defined for the native compiler.

Table 4-1 shows preprocessor symbols that are predefined when certain options are specified.

Table 4-1. Preprocessor Symbols Predefined by Options

Preprocessor Symbol	VMS Option	UNIX/DOS Option
_PIC	/code_addresses=pcrelative	-Mcp
_PID	/data_addresses=pcrelative	-Mdp
_PID	/data_addresses=a*n*relative	-Md*n*
_68000	/cpu=68000	-p68000
_68010	/cpu=68010	-p68010
_68020	/cpu=68020	-p68020
_68030	/cpu=68030	-p68030
_68040	/cpu=68040	-p68040
_CPU32	/cpu=cpu32	-pcpu32
_FPU	/fpu	-f
_MRI_EXTENSIONS	/mri_extensions	-x
_CHAR_SIGNED	/nounsignedchar	-nKu
_CHAR_UNSIGNED	/unsignedchar	-Ku

Preprocessor Directive Groups

The following types of preprocessor directives exist: the source file include directive, the line number control directive, the macro definition directives, the conditional compilation directives, and the #pragma directives.

Source File Include Directive

Text from a secondary source file can be included during compilation with the #include directive. The number of source file include directives that can be nested is limited only by the number of host system files that the process can open.

Line Number Control Directive

The line control directive #line changes the values used in reporting source file line numbers and file names.

Macro Definition Directives

Macros are defined with the #define directive and undefined with the #undef directive. Macros define names that are associated with arbitrary character sequences. When the name is encountered in the source file, it is replaced by the string of characters defined by the directive.

Macros can have parameters. When the macro is used, it is given corresponding strings to substitute for the parameters. In addition, macro definitions can be removed with the #undef directive.

The preprocessor implements macros with literal string substitution. If the macro definition is to be expanded as a single argument in an expression, it may require parentheses to get the proper order of evaluation. For example, the following definition does not contain parentheses:

```
#define nibble(i) i & 0xF
...
j = nibble(j)+1;
```

Because the definition above does not contain parentheses, the last line expands to:

```
j = j & 0xF+1;
```

which is interpreted as `j = j & (0xF + 1)` since the "+" operator has higher precedence than the "`&`" operator. If you meant:

```
j = (j & 0xF) + 1;
```

your definition of `nibble` should have been:

```
#define nibble(i) ((i) & 0xF)
```

Enclosing each individual parameter in parentheses is recommended. Rules of associativity are not always obvious, and you could obtain unexpected results. For example, the statements:

```
#define nibble(i) (i & 0xf)
nibble (j^0xff)
```

expand to:

```
(j^0xff & 0xf)
```

which will have the following associativity:

```
(j^(0xff & 0xf))
```

Therefore, the recommended definition is:

```
#define nibble(i) ((i) & 0xf)
```

The parentheses must immediately follow the macro name. If a space follows a macro name, the preprocessor will not recognize it as a macro. By convention, macro names without parameters are customarily written in uppercase, but the preprocessor will accept lower or mixed case as well.

Conditional Compilation Directives

The conditional compilation directives are #if, #ifdef, #ifndef, #else, #elif, and #endif. In the discussion of these directives, the phrase "true lines" means any source lines following an #if, #ifdef, or #ifndef directive up to the corresponding #else, #elif, or #endif (whichever comes first). The phrase "false lines" means the other set of lines (those following #else or #elif and preceding the #endif).

The conditional compilation directives allow the preprocessor to pass (or not pass) source lines to the compiler based on the value of a constant expression. These directives resemble an if statement. The conditional compilation directives begin with an #if directive followed by lines to be included when the constant expression following the directive evaluates to true. These lines are in turn followed by an optional #else part. The whole construct is terminated by an #endif directive.

Conditional compilation constructs can be nested. The following is an example of a nested conditional compilation construct.

```
#if DEBUG

#if CALLTRACING
    printf ("funcX called with %s, %d\n",par1, par2);
#endif

#else

#if SPECDEBUG
    push ("funcX");
#endif

#endif
```

In this example, if DEBUG and CALLTRACING are non-zero, a `printf` of the statement is compiled. If DEBUG is zero and SPECDEBUG is non-zero, a call to push the function name on the stack is compiled. The symbols DEBUG, CALLTRACING, and SPECDEBUG are in uppercase, indicating that they are preprocessor macros set with #define directives. Preprocessor macros are customarily written in uppercase although the compiler does not require it.

#pragma Directives

The #pragma directives are #pragma asm, #pragma endasm, #pragma list, and #pragma option, which each have specific functions. The #pragma directives allow you to include implementation-dependent information for the compiler.

Preprocessor Directives

This section provides a syntax, description, and example of each preprocessor directive provided by the Microtec Research compiler. All preprocessor directives are signaled by a pound sign (#) and must begin on a newline. Names of the directives are not subject to macro expansion before evaluation. The directives are listed in Table 4-2.

Table 4-2. Preprocessor Directives

Preprocessor Directives		
#define	#error	#include
#elif	#if	#line
#else	#ifdef	#pragma
#endif	#ifndef	#undef

Syntax Categories

The preprocessor directives recognize the following syntax categories:

error_message	*line_number*	*parameter_namelist*
expression	*macro_name*	*pragma_directive*
file_name	*option_list*	*replacement_text*
include_name		

Each of the syntax categories is described below.

Error_message

An *error_message* follows an #error directive. The error message allows you to indicate why the preprocessing has stopped.

Expression

An *expression* follows the #if and #elif directives and can use the operators listed in Table 4-3. The operators are listed in decreasing order of default precedence. You can use parentheses to override the default precedence.

Table 4-3. Expression Operators (Highest Precedence First)

Expression Operators			
defined			
+ (unary)	- (unary)	~	!
* (binary)	/	%	
+ (binary)	- (binary)		
<	<=	>	>=
==	!=		
& (binary)			
^			
|			
&&			
||			
?: (trinary)			

Note

The defined operator lets you test whether a macro name is defined. It is used only in #if and #elif constructs.

The macro name can optionally be enclosed in parentheses. If the name has been defined previously, the operator will yield a 1; if it is undefined, the operator will yield a 0.

File_name

The *file_name* optionally follows the #line directive and is expected to be a valid host system file name.

Include_name

The *include_name* follows the #include directive. The *include_name* must be a file name enclosed by double quotes or angle brackets. For example, "header.h" and <stdio.h> are valid include names.

Line_number

The *line_number* is a positive integer and follows the #line directive.

Macro_name

Macro_name follows the #ifdef, #ifndef, #define, and #undef directives and is not subject to macro expansion before evaluation. *Macro_name* can be any name that obeys the rules for C identifiers.

Option_list

The #pragma option directive allows you to specify a list of command-line options. The options given in the *option_list* must be specified using UNIX/DOS syntax; VMS syntax is not recognized.

Parameter_namelist

Parameter_namelist follows the #define directive and is not subject to macro expansion before evaluation. *Parameter_namelist* can contain any parameter name that obeys the rules for C identifiers.

Pragma_directive

The #pragma directive allows you to include implementation-dependent information for the compiler. No restrictions are placed on the information following the #pragma command, and implementations should ignore information they do not understand.

Replacement_text

Replacement_text follows the #define directive. *Replacement_text* is expanded only during macro expansion and not during macro definition.

Example:

```
#define a b
#define c a
```

In the example above, macro `a` is not replaced with `b` in the second directive. The substitution will be made later when macro `c` is expanded.

The following operators can be used within the *replacement_text* of a #define directive.

This operator has the effect of placing double quotes around the argument that follows it, thus transforming the argument into a string. This operator must be followed by a parameter name.

Example:

The # operator encloses the value of the following parameters in double quotes.

```
#define debug(x) printf(#x "=%d",x)
debug(z);
```

These statements expand to:

```
printf("z" "=%d", z);
```

Adjacent character strings are concatenated.

This operator concatenates two arguments into one token.

Example:

The ## operator splices the value of two operators together into a single token.

```
#define str(s) #s
#define tempname(n) str(temp ##n)
f=fopen(tempname(2), "w");
```

These statements expand to:

```
f=fopen("temp2", "w");
```

#define — Defines a Macro

Syntax

`#define` *macro_name* [[(*parameter_namelist*)] *replacement_text*]

Description

macro_name A user-defined name for the macro.

parameter_namelist A list of the names of macro parameters. White space is not permitted between the macro name and the left parenthesis.

replacement_text Text that is substituted for each subsequent occurrence of *macro_name* in the source program (unless undefined with #undef). The line can be continued with a backslash before the newline.

The **#define** directive defines a macro. Wherever the items in the *parameter_namelist* appear in the *replacement_text*, they are replaced by arguments given when the macro is expanded.

The arguments to the macro can be any sequence of characters. Commas that are part of an argument must be placed in parentheses. Commas that occur in generated character strings are not considered to be separate arguments.

Example

```
#define MAXCHAR 127
...
int countarray [MAXCHAR];
...
#define maximum(a,b) ((a>b)?(a):(b))
...
return (maximum (x,y));
```

Note

If a parameter to this macro has side effects (e.g., a function call), then those effects can occur twice. For example, the following:

```
maximum(i++,j--);
```

will be expanded to:

```
((i++>j--)?(i++):(j--))
```

This increments `i` twice or decrements `j` twice.

#elif — Ends the True Part of a Conditional Construct and Begins Another Conditional Construct

Syntax

`#elif` *expression*

Description

expression An expression that evaluates to true (non-zero) or false (zero).

The #elif directive provides another conditional branch for a preceding #if directive. See the #if directive for more information.

Examples

```
#if VERSION = 1
...
#elif VERSION = 2
...
#elif VERSION = 3
...
#else
#error Not Valid Version Number
#endif


#if (defined(IDENT1) && defined(IDENT2))
   i = 1;
#elif (defined(IDENT1) || defined(IDENT2))
   i = 2;
#else
   i = 3;
#endif
```

#else — Ends a Sequence of Conditional Constructs and Starts the False Part

Syntax

```
#else
```

Description

The #else directive ends the true part of a conditional compilation construct and starts the false part. The true part is passed on to the compiler when the #elif, #if, #ifdef, or #ifndef directive evaluates to true; the false part is passed on to the compiler when the #if directive evaluates to false.

Example

```
#if AMERICAN
   char vehicle[] = "truck";
#else
   char vehicle[] = "lorry";
#endif
```

#endif — Terminates a Conditional Construct

Syntax

```
#endif
```

Description

The #endif directive terminates a conditional compilation construct.

Example

```
#if AMERICAN
   char vehicle[] = "truck";
#else
   char vehicle[] = "lorry";
#endif
```

#error — Issues an Error Message then Halts Compilation

Syntax

```
#error error_message
```

Description

error_message An error message.

An error message is printed following the #error directive. No quotes are required around the error message.

Example

```
#if VERSION < 3
#error Not Valid Version Number
#endif
```

#if — Begins a Conditional Construct

Syntax

`#if` *expression*

Description

expression An expression that evaluates to true (non-zero) or false (zero).

If *expression* is true, the lines following #if are passed on to the compiler and any false lines (following #else) are discarded. If *expression* is false, the true lines (following #if) are discarded and the lines following #else are passed to the compiler.

Examples

```
#if AMERICAN
   char vehicle[] = "truck";
#else
   char vehicle[] = "lorry";
#endif


#if (defined(IDENT1) && defined(IDENT2))
   i = 1;
#elif (defined(IDENT1) || defined(IDENT2))
   i = 2;
#else
   i = 3;
#endif
```

#ifdef — Tests Whether a Macro Name is Defined

Syntax

```
#ifdef macro_name
```

Description

macro_name The name of a macro.

If *macro_name* is defined to the preprocessor, the true lines are passed on to the compiler and any false lines are discarded. If *macro_name* is undefined, the true lines are discarded and any false lines are passed on to the compiler.

To define the macro name to the preprocessor, the name must have occurred in a #define, or with the command line option described under *Define a Preprocessor Macro Name* in the *User's Guide*. Also, the macro name must not have occurred in an intervening #undef directive.

Example

```
#ifdef MAXLEN
   int line[MAXLEN];
#else
   int line[1000];
#endif
```

#ifndef — Tests Whether an Identifier is Not Defined

Syntax

```
#ifndef macro_name
```

Description

macro_name The name of a macro.

If *macro_name* is undefined to the preprocessor, the true lines are passed on to the compiler and any false lines are discarded. If *macro_name* is defined, the true lines are discarded and any false lines are passed on to the compiler.

To define the macro name to the preprocessor, the name must have occurred in a #define, or with the command line option described under *Define a Preprocessor Macro Name* in the *User's Guide*. Also, the macro name must not have occurred in an intervening **#undef** directive.

Example

```
#ifndef MAXINT
#define MAXINT 2147483647
#endif
```

#include — Declares an Include File

Syntax

```
#include "include_name"
#include <include_name>
```

Description

include_name The name of an include file.

The compiler replaces the entire #include line with the contents of *include_name* (the include file). The include file can substitute input with an #include directive up to an arbitrary number of levels. When the end of the include file is reached, text is again taken from the file containing the #include directive, resuming at the following line. Text after the file name on the #include line is ignored.

If the file name is enclosed in quotation marks (" "), the compiler will search for the include file:

1. in the directory containing your source file.
2. in the directory specified by the /ipath (VMS) or -I (UNIX/DOS) option.

If the include file is not found in either of these locations, the compiler will treat the file name as if it were enclosed in angle brackets. The compiler will then search for the include file:

1. in the directory *dir* specified by the /spath (VMS) or -J (UNIX/DOS) option.
2. in the directory specified in the environment variable MRI_68K_INC.
3. in the standard include directory if the environment variable is not defined. See the *User's Guide* for more information on the standard include directory for your host.

Examples

```
#include "mydefs.h"
#include <stdio.h>
```

Macros are also allowed:

```
#define STDIO <stdio.h>
#include STDIO
```

#line — Changes the Line Number

Syntax

`#line` *constant* [*file_name*]

Description

constant — A number that the compiler uses to number the next source line; the actual line number is not changed.

file_name — The name of the file that the compiler uses for the source file; the name of the file being read does not change.

The #line directive changes the line number (_ _LINE_ _) and, optionally, the file name (_ _FILE_ _) reported by the compiler. Error messages will be based on the new number and name.

The #line directive is used as a compiler directive and not as a preprocessor directive. It is generated by the preprocessor at the beginning and end of file inclusion or conditional selection. This allows the compiler to report errors accurately.

Example

```
#line 1000 "source_filename.c"
```

Macros are also allowed:

```
#define FIFTY 50
#define NAME "xxx.1"
#line FIFTY NAME
```

#pragma asm — Begins Embedded Assembly Instructions
#pragma endasm — Ends Embedded Assembly Instructions

Syntax

```
#pragma asm
   assembly_text
#pragma endasm
```

Description

assembly_text Assembly-language directives and statements.

The **#pragma asm** directive indicates that the following source lines are to be interpreted as assembly code. The **#pragma endasm** directive indicates the end of the assembly code sequence.

The **#pragma asm** and **#pragma endasm** directives can only be used outside of a function body. You may also use the **asm** pseudo-function to embed assembly code within a function body.

Macros and preprocessing directives can be used within the assembly code.

Example

```
#pragma options -p68010
#pragma asm
#define RET_REG d0
        XDEF    _get_cc
_get_cc:
        moveq   #0,RET_REG
        move.w  ccr,RET_REG
        rts
#pragma endasm
```

#pragma list — Suppresses Source Line Listings

Syntax

```
#pragma list off
#pragma list on
#pragma list resume
```

Description

The **#pragma list off** directive indicates that the following source lines are not to be output to the listing file regardless of the setting of the list file command line option (see *Write List File to Standard Output* in the *User's Guide*).

The **#pragma list on** directive indicates that the following source lines are to be output to the listing file. This directive has an effect only if the list file command line option is specified. For more information, see *Write List File to Standard Output* in the *User's Guide*.

The **#pragma list resume** directive restores the status of the listing to the setting that was in force prior to its previous matching **#pragma list off** or **#pragma list on** directive.

Example

To disable the listing of a segment of source code, enclose it with:

```
#pragma list off
   source_code
#pragma list resume
```

This will suppress its listing without affecting the listing status of the surrounding code.

#pragma option — Specifies Command-Line Options

Syntax

```
#pragma option [ option_list ]
```

Description

option_list Command line options specified using UNIX/DOS syntax; VMS syntax is not recognized.

The #pragma option directive allows command line options to be specified in the source file. Generally, only command line options related to optimization or code-generation can be successfully used with this pragma. Options given with this pragma have scope over the entire module.

Options for the driver or preprocessor are ignored and may cause a warning message to be given.

Example

To ensure that the module is always compiled with debugging information, use the following pragma in the source code:

```
#pragma option -g
```

#undef — Cancels the Definition of a Macro Name

Syntax

`#undef` *macro_name*

Description

macro_name The name of a macro.

The #undef directive cancels the definition of the *macro_name* by the #define directive. The *macro_name* is no longer defined for the preprocessor. It can then be redefined with a subsequent #define directive.

Example

```
#define INDENT 5
...
#undef INDENT
#define INDENT 10
...
```

Library Functions 5

The Microtec Research C Compiler provides a run-time library that contains many functions familiar to C programmers. These functions perform such operations as string manipulation and formatting, input/output, arithmetic operations, and memory management. Microtec Research functions have the same name and meaning as their UNIX C library counterparts.

Some low-level system functions are included with Microtec Research C for use on target systems; however, these functions may not be applicable to embedded systems.

Microtec Research C also provides standard include files that can be "included" in your program. These files contain declarations of run-time library functions and several standard macros. Include files are used for efficiency and reduce the risk of invoking functions and macros incorrectly.

C Function Groups and Include Files

Each library function is declared in one of the include files provided with the compiler. Include files define the names of types, constants, and external functions. They are incorporated using the #include directive and angle brackets (for example, #include <stdio.h>). The standard include files that are provided with the compiler are:

assert.h	math.h	stddef.h
ctype.h	mathf.h	stdio.h
errno.h	mriext.h	stdlib.h
float.h	setjmp.h	string.h
limits.h	signal.h	time.h
locale.h	stdarg.h	

The Microtec Research C standard include file directory is described in the *User's Guide*.

Note

Microtec Research include files may define functions which are not implemented in the libraries. This is done to have the header files conform to the ANSI C specification and to improve error detection.

The following functions, while they are defined in the header files, are not implemented in the libraries:

File	Function
locale.h	localeconv
signal.h	raise
	signal
stdio.h	freopen
	rename
	setvbuf
	tmpfile
	tmpnam
stdlib.h	getenv
	mblen
	mbstowcs
	mbtowc
	system
	wcstombs
	wctomb
string.h	strxfrm
time.h	strftime

assert.h

The standard include file **assert.h** defines the assert macro listed in Table 5-1 for putting diagnostics into programs.

Table 5-1. assert.h Macros

Macro	Description
assert	Puts diagnostics into programs

ctype.h

The standard include file **ctype.h** contains macro definitions and functions for handling characters. All functions, except **toupper** and **tolower**, are additionally defined as macros. This file should be included (using the **#include** directive) in C modules that reference the functions listed in Table 5-2.

The functions or macros with the prefix is take a character type parameter and act as predicates (they evaluate to 0 or non-zero). They can be used in **if**, **while**, and **for** constructs. The functions with the prefix to convert characters between upper and lowercase.

Table 5-2 gives the conditions under which these functions return non-zero (i.e., when the function returns true); they return 0 under all other conditions.

Table 5-2. ctype.h Functions

Function	Evaluates to 1 if Parameter Is:
isalnum (*c*)	A letter or a decimal digit
isalpha (*c*)	Alphabetic
iscntrl (*c*)	A control character: 00 through 1F hex or a delete (7F hex)
isdigit (*c*)	A decimal digit [0-9]
isgraph (*c*)	Any printing character except space
islower (*c*)	A lower-case letter
isprint (*c*)	A printable character: 20 through 7E hex
ispunct (*c*)	A punctuation character: all characters except space, digits, and letters
isspace (*c*)	A space, tab, return, newline, or form feed
isupper (*c*)	An upper-case letter
isxdigit (*c*)	A hexadecimal digit 0 to 9, A to F, or a to f
tolower (*c*)	Its argument is converted to lowercase only if it is in uppercase
toupper (*c*)	Its argument is converted to uppercase only if it is in lowercase

errno.h

The errno.h include file defines several macros listed in Table 5-3 which all relate to the reporting of error conditions.

Table 5-3. errno.h Macros

Macro	Description
EDOM	Expands to non-zero integral constant expression indicating domain errors
ERANGE	Expands to non-zero integral constant expression indicating range errors
errno	Expands to a modifiable lvalue which is set to a positive error code by several library functions

float.h

The float.h include file defines several macros that expand to various limits and parameters for floating-point types. For more information on this file, refer to *The C Programming Language, Second Edition*, by Kernighan and Ritchie.

limits.h

The limits.h include file defines several macros that expand to various limits and parameters for integral types. For more information on this file, refer to *The C Programming Language, Second Edition*, by Kernighan and Ritchie.

locale.h

The standard include file locale.h contains the information that sets the "locale" and environment for the system (see Table 5-4).

Table 5-4. locale.h Functions

Function	Definition
setlocale	Sets or queries locale information

math.h

The standard include file math.h defines the commonly-used transcendental functions in the run-time library. This file should be included (using the #include directive) in C modules that reference the C run-time library math functions.

The mathematics functions are listed in Table 5-5.

Table 5-5. math.h Functions

Function	Definition
acos (*d*)	Computes arc cosine of *d*
asin (*d*)	Computes arc sine of *d*
atan (*d*)	Computes arc tangent of *d*
atan2 (*d1*,*d2*)	Computes arc tangent of *d1*/*d2*
ceil (*d*)	Rounds *d* up to the nearest integer
cos (*d*)	Computes cosine of *d*
cosh (*d*)	Computes hyperbolic cosine of *d*
exp (*d*)	Computes e to power of *d*
fabs (*d*)	Computes absolute value of *d*
floor (*d*)	Rounds *d* down to the nearest integer
fmod (*d1*,*d2*)	Computes the remainder of *d1* divided by *d2*
frexp (*d*,*y*)	Breaks *d* into fraction and exponent
ldexp (*d*,*y*)	Computes $d * 2^y$
log (*d*)	Computes natural logarithm of *d*
log10 (*d*)	Computes base 10 logarithm of *d*
modf (*d*)	Computes the signed fractional part of *d*
pow (*d*,*y*)	Computes *d* raised to the power *y*
sin (*d*)	Computes sine of *d*
sinh (*d*)	Computes hyperbolic sine of *d*
sqrt (*d*)	Computes square root of *d*
tan (*d*)	Computes tangent of *d*
tanh (*d*)	Computes hyperbolic tangent of *d*

The math.h standard include file also defines an error macro. This macro is listed in Table 5-6.

Table 5-6. math.h Error Macros

Macro	Definition
HUGE_VAL	Result overflow (approximates largest floating-point number)

mathf.h

Under certain conditions, the standard include file math.h will include mathf.h. mathf.h declares the standard math.h functions as functions which accept single-precision (float) arguments and return float values. For more details, see the math.h file.

The functions in mathf.h are listed in Table 5-7.

Table 5-7. mathf.h Functions

Function	Definition
acosf (*f*)	Computes arc cosine of *f*
asinf (*f*)	Computes arc sine of *f*
atanf (*f*)	Computes arc tangent of *f*
atan2f (*f1*,*f2*)	Computes arc tangent of *f1*/*f2*
ceilf (*f*)	Rounds *f* up to the nearest integer
cosf (*f*)	Computes cosine of *f*
coshf (*f*)	Computes hyperbolic cosine of *f*
expf (*f*)	Computes e to power of *f*
fabsf (*f*)	Computes absolute value of *f*
floorf (*f*)	Rounds *f* down to the nearest integer
fmodf (*f1*,*f2*)	Computes the remainder of *f1* divided by *f2*
frexpf (*f*,*y*)	Breaks *f* into fraction and exponent
ldexpf (*f*,*y*)	Computes $f * 2^y$
logf (*f*)	Computes natural logarithm of *f*
log10f (*f*)	Computes base 10 logarithm of *f*
modff (*f*)	Computes the signed fractional part of *f*
powf (*f*,*y*)	Computes *f* raised to the power *y*
sinf (*f*)	Computes sine of *f*
sinhf (*f*)	Computes hyperbolic sine of *f*
sqrtf (*f*)	Computes square root of *f*
tanf (*f*)	Computes tangent of *f*
tanhf (*f*)	Computes hyperbolic tangent of *f*

The **mathf.h** standard include file also defines an error macro. This macro is listed in Table 5-8.

Table 5-8. mathf.h Error Macros

Macro	Definition
HUGE_VAL	Result overflow (approximates largest floating-point number)

mriext.h

The include file **mriext.h** includes Microtec Research extensions beyond the ANSI C function set. The other include files described in this chapter contain only ANSI features. The functions in **mriext.h** are listed in Table 5-9.

Table 5-9. mrlext.h Functions

Function	Definition
eprintf	Prints to standard error
fileno	Gets file number
ftoa	Converts floating-point numbers to ASCII
getl	Reads long int
getw	Reads short int
isascii	Checks if argument is ASCII
itoa	Converts integer to ASCII
itostr	Converts int to ASCII, variable base
ltoa	Converts long to ASCII
ltostr	Converts long to ASCII, variable base
max	Returns maximum value
memccpy	Copies memory looking for a character
memclr	Clears memory bytes
min	Returns minimum value
putl	Writes long int
putw	Writes short int
swab	Swaps bytes in memory
toascii	Guarantees argument is ASCII
_tolower (*c*)	Is the same as tolower except the argument must be in uppercase
_toupper (*c*)	Is the same as toupper except the argument must be in lowercase
zalloc	Allocates data space dynamically and initializes to zero

The mriext.h file also defines the macros listed in Table 5-10.

Table 5-10. mriext.h Macros

Macro	Definition
BLKSIZE	Is the size of I/O buffer
FALSE	Is the value 0
NULLPTR	Is the value of NULL pointer
stdaux	Is a pointer to a standard auxiliary output device
stdprn	Is a pointer to a standard printer device
TRUE	Is the value 1

setjmp.h

The standard include file setjmp.h defines two functions (setjmp and longjmp) and a type (jmp_buf) for setting up non-local jumps. The functions are listed in Table 5-11.

Table 5-11. setjmp.h Functions

Function	Definition
longjmp	Restores calling environment saved by setjmp
setjmp	Saves calling environment

The type is listed in Table 5-12.

Table 5-12. setjmp.h Types

Type	Definition
jmp_buf	Holds environment information

signal.h

The standard include file signal.h defines numerous symbolic constants and macros used to process "signals" or interrupts.

stdarg.h

The standard include file **stdarg.h** defines several macros, function prototypes, and data types for processing a variable number of arguments on a function call. These macros, functions, and data types are listed in Table 5-13 and Table 5-14.

Table 5-13. stdarg.h Functions

Function	Definition
va_arg	Accesses variable argument list
va_end	Terminates access to variable argument list
va_start	Initializes variables used by va_arg

Table 5-14. stdarg.h Types

Type	Definition
va_list	Holds information needed by va_arg, va_end, and va_start

stddef.h

The **stddef.h** include file defines some types and macros used in the description of functions. They are listed in Table 5-15 and Table 5-16, respectively.

Table 5-15. stddef.h Types

Type	Description
ptrdiff_t	Is the signed integral type which is the result of subtracting two pointers
size_t	Is the unsigned integral type which is the result of the sizeof operator
wchar_t	Is the initialized type whose range of values can represent distinct codes for all numbers of the character set supported by the compiler

Table 5-16. stddef.h Macros

Macro	Description
NULL	Expands to a null pointer constant
offset (*type*, *identifier*)	Expands to an integral constant expression that has type size_t, the value of which is the offset in bytes of the structure member *identifier* from the beginning of its structure type

stdio.h

The standard include file **stdio.h** contains external declarations for I/O functions and formatted I/O functions. It also contains type declarations and macro definitions for several I/O related functions such as **clearerr**, **feof**, **ferror**, **getc**, **getchar**, **putc**, and **putchar**.

The standard include file **stdio.h** must be included to use these I/O functions, macros, and symbol definitions.

The I/O functions take **stdin**, **stdout**, **stderr**, or another file variable as an argument. However, in embedded systems, file arguments can be ignored since neither file opening nor file closing mechanisms are typically available on an embedded system. In that case, all input is connected to the keyboard; all output is connected to the screen. The names are used only as place holders for portability. The I/O functions are listed in Table 5-17.

Table 5-17. stdio.h I/O Functions

Function	Definition
clearerr	Clears error or End-Of-File condition
fclose	Closes a specified file
feof	Indicates if a file is at End-Of-File
ferror	Indicates if an error condition has occurred
fflush	Flushes any buffered data to a file
fgetc	Gets a character from a specified file
fgetpos	Gets the current location in a file
fgets	Reads a string from a specified file
fopen	Opens a file
fputc	Adds a character to a specified file
fputs	Puts a string into the specified file
fread	Reads from a specified file
fseek	Sets the current location in a file
fsetpos	Sets the current location in a file
ftell	Gets the current location in a file
fwrite	Writes to a specified file
getc	Reads a character from a file
getchar	Reads a character from the standard input device
gets	Reads a line from standard input device
perror	Writes an error message to the standard error device
putc	Writes a character to a file
putchar	Writes a character to the standard output device
puts	Writes a string to standard output
remove	Makes file name inaccessible
rewind	Points to the beginning of a file
setbuf	Associates a buffer with an I/O file
ungetc	Returns a character to a file

Since embedded systems do no usually have a file system, some functions are provided with a stub only; they simply return 0. If they are used, they will generate a link-time error indicating that the symbol _WARNING_*xxx*_stub_used is unresolved. The string *xxx* will be replaced by the function name. If they are executed, they will write "WARNING-*xxx* (stub routine) called" to stderr.

The formatted I/O functions perform formatted I/O. The standard I/O functions scanf and printf operate on stdin and stdout implicitly. The formatted I/O functions are defined in Table 5-18. The ANSI-portable macros and structures are shown in Table 5-19.

Table 5-18. stdio.h Formatted I/O Functions

Function	Definition
fprintf	Performs formatted print to specified file
fscanf	Performs formatted read from specified file
printf	Performs formatted print to standard output device (stdout)
scanf	Performs formatted read from standard input device (stdin)
sprintf	Performs formatted print to memory
sscanf	Performs formatted read from memory
vfprintf	Performs formatted print to a file
vprintf	Performs formatted print to standard output device
vsprintf	Formats a string

Table 5-19. Macros and Structures

Symbol	Definition
BUFSIZ	This macro expands to an integer constant expression, which is the size in bytes of the buffer used by an input/output stream.
EOF	This symbol is compared with the values returned by library input routines (such as getc and putc) to signal End-Of-File or an error condition. The information includes a file pointer, a pointer to an associated buffer, and an indicator that detects End-Of-File and records read/write errors that may have occurred.
FILE	This structure type is capable of recording information needed to control an input/output stream.
FILENAME_MAX	This macro indicates the size needed for a character array large enough to hold the longest file name string that the compiler guarantees can be opened. As the compiler imposes no practical limit on the length of file name strings, FILENAME_MAX is the recommended size of an array intended to hold a file name string. This value may be changed by the programmer.
FOPEN_MAX	This macro indicates the guaranteed minimum number of file names that can be opened simultaneously.
L_tmpnam	This macro indicates the size of a character array that is large enough to hold a temporary file name string.
NULL	This macro indicates a null pointer constant (value is 0).
SEEK_CUR	This macro indicates the current position in a file.
SEEK_END	This macro indicates an End-Of-File position in a file.
SEEK_SET	This macro indicates the beginning position in a file.
stderr	This macro points to a file object associated with the standard error device.
stdin	This macro points to a file object associated with the standard input device.
stdout	This macro points to a file object associated with the standard output device.
TMP_MAX	This macro specifies the minimum number of unique file names that can be generated by the tmpnam function.

stdlib.h

The standard include file **stdlib.h** contains external declarations for format conversion and general utility functions. The format conversion functions convert data between integer and ASCII format. The ASCII input form can be either decimal or hexadecimal. The memory management functions manage dynamically allocated data storage. The general utilities functions are listed in Table 5-20.

Table 5-20. stdlib.h Functions

Function	Definition
abort	Causes abnormal program termination
abs	Computes absolute value of an integer
atexit	Queues functions for execution at termination
atof	Converts ASCII to floating point
atoi	Converts ASCII to integer
atol	Converts ASCII to long integer
bsearch	Performs binary search
calloc	Allocates data space dynamically
div	Computes quotient and remainder
exit	Closes all opened files and terminates program
free	Frees dynamically allocated data space
labs	Computes absolute value of a long integer
ldiv	Computes quotient and remainder
malloc	Allocates data space dynamically
qsort	Is a quick sort algorithm
rand	Returns a positive pseudo-random number
realloc	Changes block size of allocated space
srand	Sets the seed of the random number generator
strtod	Converts ASCII to double
strtol	Converts ASCII to a long with specified base
strtoul	Converts string to unsigned long integer

string.h

The standard include file string.h contains declarations for all C run-time library string (str) and memory (mem) functions. This file should be included (using the #include directive) in C modules that reference the C run-time library string or memory functions.

This file also declares the type size_t and the macro NULL which are described for stddef.h.

The string functions copy, append, and test character strings in the normal C character string format (i.e., terminated with a NULL character). The string functions are listed in Table 5-21.

Table 5-21. string.h String Functions

Function	Definition
strcat	Concatenates two strings
strchr	Locates first occurrence of char in string
strcmp	Compares two strings
strcoll	Compares two strings
strcpy	Copies a string
strcspn	Computes initial segment length of string
strerror	Points to an error message
strlen	Determines length of a string
strncat	Performs a limited concatenation of two strings
strncmp	Performs a limited comparison of two strings
strncpy	Performs limited copying of a string
strpbrk	Finds break character in string
strrchr	Locates last occurrence of char in string
strspn	Computes initial segment length of string
strstr	Finds first occurrence of characters in string
strtok	Locates tokens in a string

The memory functions copy, test, and set non-NULL terminated byte sequences in memory. The memory functions are listed in Table 5-22.

Table 5-22. string.h Memory Functions

Function	Definition
memchr	Searches for characters in memory
memcmp	Compares arguments in memory
memcpy	Copies characters from memory
memmove	Copies characters from memory
memset	Sets values of characters in memory

time.h

The standard include file time.h defines a structure (tm) and several functions for manipulating representations of time. The time and date functions are listed in Table 5-23.

Table 5-23. time.h Functions

Function	Definition
asctime	Converts time into string format
clock	Returns processor time used
ctime	Converts calendar time into local time in string format
difftime	Computes difference between two calendar times
gmtime	Converts calendar time into Greenwich Mean Time (GMT)
localtime	Converts calendar time into local time
time	Returns calendar time

System Functions

The system functions that perform input/output operations are defined in Table 5-24.

Table 5-24. System Functions

Function	Definition
close	Closes a specified file
creat	Creates a new file
_exit	Terminates a program
lseek	Sets the current location in a file
open	Opens a specified file
read	Reads from a specified file
sbrk	Allocates memory space
unlink	Unlinks a file name
write	Writes to a specified file

Since embedded systems do not usually have a file system, the **close**, **lseek**, **open**, and **unlink** functions are provided with stubs only; they simply return 0. If **lseek**, **open**, or **unlink** are used, they will generate a link-time error indicating that the symbol _WARNING_*xxx*_stub_used is unresolved. The string *xxx* will be replaced by the function name. If it is executed, it will write "WARNING - *xxx* (stub routine) called" to **stderr**.

The **read** function reads from the variable **_simulated_input**. The **write** function writes to the variable **_simulated_output**. If you want to replace any of these functions with one of your own, link in your implementation before the run-time library.

Non-Reentrant Functions

Microtec Research libraries are as reentrant as possible. However, ANSI C defines some library functions in such a way that it is impossible to implement them in a reentrant fashion. If a library function is defined so that it uses static, non-automatic data, then the function is non-reentrant.

The largest class of functions which accesses static data are those routines in stdio.h. In general, if reentrancy is required, these routines should be avoided. Three notable exceptions are sprintf, vsprintf, and sscanf, which are completely reentrant.

Functions which modify errno are also non-reentrant. Many of the functions declared in math.h modify the static variable errno, along with the functions strtod, strtol, strtoul, and others. A list of other non-reentrant functions is shown in Table 5-25.

Table 5-25. Other Non-Reentrant Functions

Function	Static Variables Used
asctime	_ctbuf[ ]
atexit	_atexit_stack[ ], _atexit_top
calloc	_membase, _avail, _badlist
ctime	_ctbuf[ ], _xtm
free	_membase, _avail, _badlist
gmtime	_xtm
localtime	_xtm
malloc	_membase, _avail, _badlist
rand	_randx
realloc	_membase, _avail, _badlist
srand	_randx
strtok	_lastp
zalloc	_membase, _avail, _badlist

Library Function Definitions

This section describes all of the library functions in alphabetical order and presents them in a fixed format for easy reference. The format used is described below.

Function Name

Each section begins with the name of the library function followed by a short, one line description. Note that some functions are also defined as macros.

Syntax

The syntax shows the return type of the function and the types of its formal arguments. Declarations for some of these functions are in the standard include file indicated. Alternatively, for functions which are not implemented as macros, the function can be declared as an external function with the appropriate return type.

Class

Each function is flagged as one of three classes: a standard C function (present in most C implementations), a system function, or a Microtec Research C extension.

Description

The description gives the meaning of the function and its return values.

Notes

This section contains information on abnormal behavior, embedded environment considerations, and other issues that may affect the behavior of the function or macro.

abort — Causes Abnormal Program Termination

Syntax

```
#include <stdlib.h>
void abort (void);
```

Class

Standard C Function

Description

The abort function causes an abnormal termination to occur. Open output streams are flushed and closed. This function does not return.

Notes

None.

abs — Computes Absolute Value of an Integer

Syntax

```
#include <stdlib.h>
int abs (int i);
```

Class

Standard C Function

Description

The abs function returns the absolute value of *i*. If *i* is less than zero, `abs (i)` equals *-i*. If *i* is greater than or equal to zero, `abs (i)` equals *i*.

Notes

The abs function returns an incorrect result when *i* is -2147483648 for target machines with 32-bit integers and when *i* is -32768 for target machines with 16-bit integers.

acos — Computes Arc Cosine of a Number

Syntax

```
#include <math.h>
double acos (double d);
```

Class

Standard C Function

Description

The acos function returns the arc cosine of *d*. The range of *d* is [-1, 1]. The range of the return value is [0, pi] radians.

If *d* is out of range, the result will be NaN and errno will be set to EDOM.

Notes

The function acos can modify the static variable errno; no other static data is modified.

For information on the single-precision version of this function, acosf, refer to the *mathf.h* section at the beginning of this chapter.

asctime — Converts Date and Time to String

Syntax

```
#include <time.h>
char *asctime (const struct tm *tptr);
```

Class

Standard C Function

Description

The asctime function converts the broken-down time in the structure pointed to by *tptr* to a 26-character string in the form:

Mon Oct 12 10:25:02 1987\n\0

The asctime function returns a pointer to the string. All the fields in the string have constant width.

Notes

The converted string is in a static data area which is overwritten by each call.

The function asctime modifies the static array **_ctbuf**; no other static data is modified.

asin — Computes Arc Sine of a Number

Syntax

```
#include <math.h>
double asin (double d);
```

Class

Standard C Function

Description

The asin function returns the arc sine of *d*. The range of *d* is [-1, 1]. The range of the return value is [-pi/2, pi/2] radians.

If *d* is out of range, the result will be NaN and `errno` will be set to EDOM.

Notes

The asin function can set the static variable `errno`; no other static data is modified.

For information on the single-precision version of this function, asinf, refer to the *mathf.h* section at the beginning of this chapter.

assert — Tests Program Expressions

Syntax

```
#include <assert.h>
void assert (int expression);
```

Class

Standard C Macro

Description

The assert macro indicates that *expression* is expected to be true at this point in the program. It prints a diagnostic message and aborts the program if *expression* is 0.

Diagnostics inserted using assert can be disabled by setting the NDEBUG macro to 1 before using the #include <assert.h> directive.

Notes

None.

atan — Computes Arc Tangent of a Number

Syntax

```
#include <math.h>
double atan (double d);
```

Class

Standard C Function

Description

The atan function returns the arc tangent of *d*. The range of the return value is [-pi/2, pi/2] radians.

Notes

The atan function can modify the static variable errno; no other static data is modified.

For information on the single-precision version of this function, atanf, refer to the *mathf.h* section at the beginning of this chapter.

atan2 — Computes Arc Tangent d1/d2 (with Overflow Protection)

Syntax

```
#include <math.h>
double atan2 (double d1, double d2);
```

Class

Standard C Function

Description

The atan2 function returns the arc tangent of *d1/d2*. The range of the return value is [-pi, pi] radians.

If both arguments are zero, the result will be zero and the error will be set to EDOM.

Notes

The function atan2 can modify the static variable errno; no other static variables are modified.

For information on the single-precision version of this function, **atan2f**, refer to the *mathf.h* section at the beginning of this chapter.

atexit — Queues Function to be Called at Program Termination

Syntax

```
#include <stdlib.h>
int atexit (void (*func)(void));
```

Class

Standard C Function

Description

The atexit function queues the function pointed to by *func*, to be called without arguments at normal program termination. The queued functions are called in reverse order. The atexit function can queue 32 function calls.

This function returns zero if it succeeds, and non-zero if it fails.

Notes

None.

atof — Converts ASCII String to Floating-Point Number

Syntax

```
#include <stdlib.h>
double atof (const char *cp);
```

Class

Standard C Function

Description

The atof function returns the double precision, floating-point value of *cp*. The pointer *cp* must point to a string. The atof function takes ASCII decimal digits from the left. The format for the string is:

[*ws*] [*sign*] *digits* ["." *digits*] [*exp* [*sign*] *digits*]

where:

ws	is one or more blanks or tabs.
sign	is a - or +.
digits	is one or more decimal digits.
exp	is e or E.

Any character other than those specified, in particular a NULL character, terminates the resulting number.

Notes

If the number represented by the string is out of range for double precision, the result is undefined.

atoi — Converts ASCII String to Integer

Syntax

```
#include <stdlib.h>
int atoi (const char *cp);
```

Class

Standard C Function

Description

The atoi function returns the signed integer value of a string. The pointer *cp* must point to a NULL-terminated string. This function takes ASCII decimal digits from the left. It ignores leading blanks and tabs. An optional plus or minus sign (i.e., + or -) can be included to obtain a signed result. Any non-digit, in particular a NULL character, terminates the number.

Notes

If the resulting number represented by the string is out of range for a signed integer, the result is undefined.

atol — Converts ASCII String to Long Integer

Syntax

```
#include <stdlib.h>
long atol (const char *cp);
```

Class

Standard C Function

Description

The atol function returns the signed long integer value of a string. The pointer *cp* must point to a NULL-terminated string. This function takes ASCII decimal digits from the left. It ignores leading blanks and tabs. You can include an optional sign (+ or -) to obtain a signed result. Any non-digit, in particular a NULL character, terminates the number.

Notes

If the resulting number represented by the string is out of range for a signed long integer, the result is undefined.

bsearch — Performs Binary Search

Syntax

```
#include <stdlib.h>
void *bsearch (const void *key, const void *base, size_t nmemb,
        size_t size, int (*compar)(const void *, const void *));
```

Class

Standard C Function

Description

The bsearch function searches a sorted array for an element which matches the object pointed to by *key*. The bsearch function uses a binary search algorithm to perform the search. The address of the initial element of the array is given by *base*; this array must have been previously sorted according the function pointed to by *compar*. The number of elements in the array is *nmemb*. The size of each element of the array is *size*.

The bsearch function calls the comparison function pointed to by *compar*. This function is called with two arguments; the first is the pointer *key* and the second is a pointer to an array element. The comparison function must return an integer less than, equal to, or greater than zero if the object pointed to by *key* is considered to be less than, equal to, or greater than the array element.

The bsearch function returns the address of a matching element of the array, or NULL if no match is found.

Notes

If two or more elements of the array compare as equal, which of the equal elements is matched is undefined.

calloc — Allocates Data Space Dynamically

Syntax

```
#include <stdlib.h>
void *calloc (size_t nelem, size_t elsize);
```

Class

Standard C Function

Description

The calloc function allocates a new area of memory space for *nelem* objects, each of whose size is *elsize* bytes; it returns a pointer to that area. The size of the area allocated is equal to *nelem* multiplied by *elsize* bytes. The calloc function sets the area to zeros. The space allocated is guaranteed to start on a boundary that is suitable for aligning any type.

If the requested amount of space cannot be allocated, calloc returns a NULL pointer.

Notes

The function calloc can modify the static struct _membase, or the static data _avail or _badlist; no other static data is modified.

ceil — Computes Integer Ceiling of Double

Syntax

```
#include <math.h>
double ceil (double d);
```

Class

Standard C Function

Description

The ceil function returns the smallest integer that is not less than *d*. The result is expressed as a double.

Notes

For information on the single-precision version of this function, ceilf, refer to the *mathf.h* section at the beginning of this chapter.

clearerr — Clears the File I/O Error Indicator

Syntax

```
#include <stdio.h>
void clearerr (FILE *file);
```

Class

Standard C Function (implemented as a function and as a macro)

Description

The clearerr function resets the error and EOF indicators on *file*.

The macro is invoked by default. If you want to call the function, you must either:

- enclose the name in parentheses:

```
(clearerr) (file);
```

or

- undefine the `clearerr` macro with `#UNDEF` before using it:

```
#UNDEF clearerr
```

Notes

The clearerr function cannot detect exceptions.

clock — Returns Processor Time Used

Syntax

```
#include <time.h>
clock_t clock (void);
```

Class

Standard C Function

Description

The function clock returns a value which may be used to compute the processor time used by a program. To do this task, the clock function should be called at the start of the program and its return value subtracted from the value returned by subsequent calls. In order to compute the time used in seconds, this difference must be divided by the macro CLOCKS_PER_SEC.

This implementation of clock only returns (clock_t)-1 to indicate that the processor time is unavailable. No usable timing values are returned.

Notes

This routine is provided as a service to the user only. If you wish to provide your own clock function, you must link it in ahead of the libraries.

close — Closes a Specified File

Syntax

```
int close (int fd);
```

Class

System Function

Description

The close function closes a file previously opened by either the open or creat function. The file descriptor *fd* is returned by open or creat when the associated file is opened. A file descriptor must be specified (as opposed to a stream address).

The close function returns 0 when successful, or -1 if the function detects an unknown file descriptor.

Notes

Since embedded systems do not usually have a file system, the close function is provided as a minimal implementation. If you are developing code for an environment that does have a file system, you can provide a replacement for the close function by linking in your implementation of close before the run-time library.

cos — Computes Cosine of a Number

Syntax

```
#include <math.h>
double cos (double d);
```

Class

Standard C Function

Description

The cos function returns the cosine of *d*. Arguments must be in radians.

Notes

A large magnitude argument can yield a result of little or no significance.

For information on the single-precision version of this function, cosf, refer to the *mathf.h* section at the beginning of this chapter.

cosh — Computes Hyperbolic Cosine of a Number

Syntax

```
#include <math.h>
double cosh (double d);
```

Class

Standard C Function

Description

The cosh function returns the hyperbolic cosine of *d*.

The cosh function returns HUGE_VAL when the correct value cannot be represented and errno is set to ERANGE.

Notes

The cosh function can modify the static variable errno; no other static variables are modified.

For information on the single-precision version of this function, coshf, refer to the *mathf.h* section at the beginning of this chapter.

creat — Creates a Specified File

Syntax

```
int creat (char *filename, int mode);
```

Class

System Function

Description

The creat function creates and opens a file for writing. The file is specified by *filename*. Its protection mode is specified by *mode*. For devices that cannot be created (such as terminals), the creat function just opens the device.

If creat is successful, it returns a file descriptor for *filename*. It returns -1 if the file cannot be written to, if the file is a directory, or if there are already too many files opened.

Notes

The creat function uses the open function which is implemented as a stub. You must provide your own open function to allow creat to work in your environment.

ctime — Converts Date and Time to a String

Syntax

```
#include <time.h>
char *ctime (const time_t *timer);
```

Class

Standard C Function

Description

The ctime function converts the calendar time, pointed to by *timer*, to local time. It returns a pointer to a 26-character string in the form:

Mon Oct 12 10:25:02 1987\n\0

The calendar time (a long integer) is in seconds and should start at 00:00:00 (local time), January 1, 1970. All the fields in the string have constant width. The ctime function is equivalent to:

```
asctime(localtime(timer))
```

Notes

The converted string is in a static data area whose content is overwritten by each call. The function does not account for Daylight Savings Time, which is dependent on the target operating system. This function is similar but not fully compatible with the UNIX implementation.

The function ctime modifies the static array _ctbuf and the static struct _xtm; no other static variables are modified.

difftime — Computes Difference Between Two Calendar Times

Syntax

```
#include <time.h>
double difftime (time_t time1 time_t time0);
```

Class

Standard C Function

Description

The **difftime** function computes the difference between two calendar times: *time1* - *time0*.

The **difftime** function returns the difference expressed in seconds as a `double`.

Notes

None.

div — Computes Quotient and Remainder for a Division

Syntax

```
#include <stdlib.h>
div_t div (int numer, int denom);
```

Class

Standard C Function

Description

The div function computes the quotient and remainder of the division of the numerator *numer* by the denominator *denom*. If the division is inexact, the resulting quotient is the integer of lesser magnitude that is the nearest to the algebraic quotient. For example:

```
quot * denom + rem = numer
```

The div function returns a structure of type `div_t`. The structure contains the following members:

```
int quot;
int rem;
```

Notes

If the result cannot be represented, the behavior is undefined.

eprintf — Provides Formatted Print to Standard Error

Syntax

```
#include <mriext.h>
int eprintf (const char *format, ...);
```

Class

Microtec Research C Extension

Description

The **eprintf** function provides formatted output to the C standard error file. The format and results are the same as for **printf**. When **eprintf** is successful, it returns the number of characters transmitted.

If an output error is encountered, **eprintf** returns **EOF** (**End-Of-File**).

Notes

The function **eprintf** modifies the static array **_iob**; no other static variables are modified.

exit — Closes All Opened Files and Terminates Program

Syntax

```
#include <stdlib.h>
void exit (int status);
```

Class

Standard C Function

Description

The exit function causes normal program termination to occur with a status code specified by *status*. When exit terminates a program, it executes enqueued atexit() function calls, flushes all opened files, closes them, then returns control to the host operating system environment. If the target system does not have a file system, flushing is not applicable.

The exit function is always called implicitly at the end of a C main() function.

Notes

A call to exit will never return.

_exit — Terminates a Program

Syntax

```
void _exit (int status);
```

Class

System Function

Description

The _exit function causes normal program termination to occur with a status code specified by *status*.

The _exit function is always called implicitly at the end of a exit( ) function call.

Notes

A call to _exit will never return.

exp — Computes Exponential Value

Syntax

```
#include <math.h>
double exp (double d);
```

Class

Standard C Function

Description

The exp function computes the exponential function of *d*. It returns the exponential value.

If the magnitude of *d* is too large, errno is set to ERANGE and the function returns HUGE_VAL.

Notes

The function exp can modify the static variable errno; no other static variables are modified.

For information on the single-precision version of this function, expf, refer to the *mathf.h* section at the beginning of this chapter.

fabs — Computes Absolute Value of a Floating-Point Number

Syntax

```
#include <math.h>
double fabs (double d);
```

Class

Standard C Function

Description

The fabs function returns the absolute value of *d*. If *d* is less than zero, `fabs(d)` equals *-d*. If *d* is greater than or equal to zero, `fabs(d)` equals *d*.

Notes

For information on the single-precision version of this function, fabsf, refer to the *mathf.h* section at the beginning of this chapter.

fclose — Closes a File

Syntax

```
#include <stdio.h>
int fclose (FILE *file);
```

Class

Standard C Function

Description

The fclose function causes any unwritten buffered data for the named *file* to be written out and the *file* to be closed.

The fclose function returns zero if the file was successfully closed, or returns non-zero if any errors were detected or if the file was already closed.

Notes

The fclose function modifies the static array _iob; no other static data is modified.

This function also calls the system function close.

feof — Checks for End-Of-File on a File

Syntax

```
#include <stdio.h>
int feof (FILE *file);
```

Class

Standard C Function (implemented as a function and as a macro)

Description

The feof function or macro checks for the End-Of-File on *file*.

The macro is invoked by default. If you want to call the function, you must either:

- enclose the name in parentheses:

  ```
  (feof) (file);
  ```

or

- undefine the `feof` macro with `#UNDEF` before using it:

  ```
  #UNDEF feof
  ```

The feof function returns non-zero when End-Of-File was previously detected reading the named input *file*. Otherwise, feof returns zero.

Notes

None.

ferror — Checks the File I/O Error Indicator

Syntax

```
#include <stdio.h>
int ferror (FILE *file);
```

Class

Standard C Function (implemented as a function and as a macro)

Description

The ferror function or macro checks the file I/O error indicator.

The macro is invoked by default. If you want to call the function, you must either:

- enclose the name in parentheses:

```
(ferror) (file);
```

or

- undefine the `ferror` macro with `#UNDEF` before using it:

```
#UNDEF ferror
```

The ferror function returns non-zero when an I/O error previously occurred when reading from or writing to the named *file*. Otherwise, ferror returns zero.

Notes

None.

fflush — Flushes Buffered Data to a File

Syntax

```
#include <stdio.h>
int fflush (FILE *file);
```

Class

Standard C Function

Description

The **fflush** function is used by fclose to flush a *file*. **fflush** causes unwritten data stored in file buffers to be written to a file.

The **fflush** function returns non-zero on error, or zero if no errors occur. This function does not write to files that are not opened for writing.

Notes

The **fflush** function modifies the static array _iob; no other static data is modified.

fgetc — Reads a Character from a Specified File

Syntax

```
#include <stdio.h>
int fgetc (FILE *file);
```

Class

Standard C Function

Description

The fgetc function reads the next character (if present) as an unsigned char converted to an int from the input *file*. It also increments the associated file pointer, if defined, by one byte. The fgetc function performs the same task as getc; however, getc is implemented as a macro. The fgetc function should be used when side effects of macro processing of arguments are a problem.

If the file is at the End-Of-File or if a read error occurs, fgetc returns EOF.

Notes

The function fgetc modifies the static array _iob; no other static data is modified.

fgetpos — Gets the Current Location in a File

Syntax

```
#include <stdio.h>
int fgetpos (FILE *stream, fpos_t *pos);
```

Class

Standard C Function

Description

The fgetpos function stores the current value of the file position indicator for *stream* in the object pointed to by *pos*. The value stored contains information usable by the fsetpos function for repositioning the stream to its position at the time of the call to the fgetpos function.

If successful, the fgetpos function returns zero; on failure, the fgetpos function returns non-zero and stores an appropriate error indicator in errno.

Notes

The fgetpos function utilizes the ftell function which is implemented as a stub. You must provide your own ftell function to allow fgetpos to work in your environment.

fgets — Reads a String from a Specified File

Syntax

```
#include <stdio.h>
char *fgets (char *cp, int i,FILE *file);
```

Class

Standard C Function

Description

The fgets function reads the number of characters specified by *i*-1 from *file* to the area pointed to by *cp*. No additional characters are read after a newline (which is retained) or after the End-Of-File. A NULL character is written immediately after the last character is read into the array.

The fgets function returns *cp* if successful, and a NULL pointer on End-Of-File or error.

Notes

The fgets function modifies the static array _iob; no other static data is modified.

fileno — Gets File Descriptor

Syntax

```
#include <mriext.h>
char fileno (FILE * stream);
```

Class

Microtec Research C Extension (implemented as a macro)

Description

The fileno macro returns the file descriptor associated with *stream*.

Notes

The macro will yield undefined results if *stream* does not point to an open file.

floor — Rounds a Floating-Point Number to Nearest Integer

Syntax

```
#include <math.h>
double floor (double d);
```

Class

Standard C Function

Description

The floor function returns a floating-point value representing the largest integer that is not greater than *d*.

Notes

For information on the single-precision version of this function, floorf, refer to the *mathf.h* section at the beginning of this chapter.

fmod — Computes Floating-Point Remainder

Syntax

```
#include <math.h>
double fmod (double d1, double d2);
```

Class

Standard C Function

Description

The fmod function calculates the floating-point remainder of *d1*/*d2*, such that `f=d1-i*d2`. In this equation, `i` is an integer, `f` has the same sign as *d1*, and the absolute value of `f` is less than the absolute value of *d2*.

If *d2* is zero or if *d1*/*d2* causes an overflow, the function returns a NaN and sets errno to EDOM.

Notes

The function fmod can modify the static variable errno; no other static data is modified.

For information on the single-precision version of this function, fmodf, refer to the *mathf.h* section at the beginning of this chapter.

fopen — Opens a File

Syntax

```
#include <stdio.h>
FILE *fopen (const char *filename, const char *type);
```

Class

Standard C Function

Description

The fopen function opens the file named by *filename* and associates a FILE structure with it. The FILE structure contains the data necessary to perform I/O on the file. In the syntax above, *type* is one of the following character strings:

Type	Meaning
r	Opens a text file for reading
w	Creates a text file for writing, or truncates it to zero length
a	Appends; opens a text file or creates one for writing at the End-Of-File
rb	Opens a binary file for reading
wb	Creates a binary file for writing, or truncates it to zero length
ab	Appends; opens a binary file or creates one for writing at the End-Of-File
r+	Opens a text file for updating (reading and writing)
w+	Creates a text file for updating, or truncates it to zero length
a+	Appends; opens a text file or creates one for updating and writing at the End-Of-File
r+b or rb+	Opens a binary file for updating (reading and writing)
w+b or wb+	Creates a binary file for updating, or truncates it to zero length
a+b or ab+	Appends; opens a binary file or creates one for updating and writing at the End-Of-File

The fopen function returns a pointer to the FILE structure associated with *file*. It returns a NULL pointer if *filename* cannot be opened.

Notes

The **fopen** function modifies the static array **_iob**; no other static data is modified.

This function also uses the **open** function which is implemented as a stub. You must provide your own **open** function to allow **fopen** to work in your environment.

fprintf — Provides Formatted Print to a Specified File

Syntax

```
#include <stdio.h>
int fprintf (FILE *file, const char format,...);
```

Class

Standard C Function

Description

The **fprintf** function provides formatted output to the file associated with *file*. The format and results are the same as for **printf**.

When **fprintf** is successful, it returns the number of characters transmitted. If an output error is encountered, **fprintf** returns a negative value.

Notes

The **fprintf** function can modify the static array **_iob**; no other static data is modified.

fputc — Writes a Character to a Specified File

Syntax

```
#include <stdio.h>
int fputc (int c, FILE *file);
```

Class

Standard C Function

Description

The fputc function writes the character *c* to the file associated with *file*. It performs the same task as putc; however, fputc is implemented as a function, whereas putc is implemented as a macro. The fputc function should be used when side effects of macro processing of arguments are a problem.

When fputc is successful, it returns the character written. If fputc fails, it returns an EOF (End-Of-File).

Notes

The function fputc can modify the static array _iob; no other static data is modified.

fputs — Writes a String to a Specified File

Syntax

```
#include <stdio.h>
int fputs (const char *cp, FILE *file);
```

Class

Standard C Function

Description

The fputs function writes the NULL-terminated string *cp* to the file associated with *file*. It does not add a new-line character to the output. The terminating NULL character is not written.

The fputs function returns non-zero if an error is detected; otherwise, it returns zero.

Notes

The function fputs can modify the static array _iob; no other static data is modified.

fread — Reads from a Specified File

Syntax

```
#include <stdio.h>
size_t fread (void *ptr, size_t size, size_t nitems, FILE *file);
```

Class

Standard C Function

Description

The **fread** function reads *nitems* (items of data) from the specified file and writes them to the memory pointed to by *ptr*. The number of bytes in each item is specified by *size*.

The **fread** function returns the number of items read, which may be less than *nitems* if a read error or End-Of-File is encountered.

Notes

The function **fread** can modify the static array _iob; no other static data is modified.

free — Frees Dynamically Allocated Data Space

Syntax

```
#include <stdlib.h>
void free (void *cp)
```

Class

Standard C Function

Description

The free function deallocates space that has been allocated by one of the allocation routines (see calloc, malloc, realloc, or zalloc). The pointer *cp* must be a pointer returned previously by an allocation routine.

Notes

The free function cannot detect exceptions. If more data is copied into the area allocated than it can hold, the heap could be corrupted. This can result in unpredictable behavior at runtime.

The function free can modify the static structure _membase, or data _avail or _badlist; no other data is modified.

frexp — Breaks Down Floating-Point Number into Mantissa and Exponent

Syntax

```
#include <math.h>
double frexp (double d, int *expptr);
```

Class

Standard C Function

Description

The frexp function breaks down the floating-point value *d* into a mantissa m and an exponent n such that the absolute value of m is greater than or equal to 0.5 and less than 1.0 and $d = m \bullet (2^n)$. The integer exponent n is stored at the location pointed to by *expptr*. The frexp function returns the mantissa m.

If *d* is zero, the function returns 0 for both the mantissa and exponent. If *d* is either NaN or infinity, frexp returns NaN and sets errno to EDOM.

Notes

The function frexp can modify the static variable errno; no other static data is modified.

For information on the single-precision version of this function, frexpf, refer to the *mathf.h* section at the beginning of this chapter.

fscanf — Reads from a Specified File

Syntax

```
#include <stdio.h>
int fscanf (FILE *file, const char *format,...);
```

Class

Standard C Function

Description

The fscanf function reads from the named input *file*. It reads characters, interprets them according to *format*, and stores the results in the remaining arguments. As arguments, it expects a control string format and a set of pointer arguments indicating where the converted input should be. The format and results for this function are the same as for scanf.

The fscanf function returns the number of successfully deposited input arguments. If an error or unexpected End-Of-Input occurs, EOF (End-Of-File) is returned.

Notes

The function fscanf can modify the static array _iob; no other static data is modified.

fseek — Sets the Current Location in a File

Syntax

```
#include <stdio.h>
int fseek (FILE *stream, long int offset, int whence);
```

Class

Standard C Function

Description

The fseek function sets the file position for the stream pointed to by *stream*. The new position is obtained by adding *offset* to the position specified by *whence*. The specified position is the beginning of the file if *whence* is SEEK_SET, the current value of the file position indicator if SEEK_CUR, or End-Of-File if SEEK_END.

The fseek function returns non-zero only for a request that cannot be satisfied.

Notes

This particular version of fseek calls lseek to perform the file positioning. The function lseek is implemented only as a stub in these libraries.

The function fseek, when fully implemented, can modify the static array _iob; no other static data is modified.

fsetpos — Sets the Current Location in a File

Syntax

```
#include <stdio.h>
int fsetpos (FILE *stream, const fpos_t *pos);
```

Class

Standard C Function

Description

The fsetpos function sets the file position indicator for *stream* according to the value of the object pointed to by *pos*, which shall be a value obtained from an earlier call to the fgetpos function on the same stream.

If successful, the fsetpos function returns zero; on failure, the fsetpos function returns non-zero and stores an appropriate error indicator in errno.

Notes

The fsetpos function utilizes the lseek function which is implemented as a stub. You must provide your own lseek function to allow fsetpos to work in your environment.

The function fsetpos can modify the static array _iob; no other static data is modified.

ftell — Gets the Current Location in a File

Syntax

```
#include <stdio.h>
long int ftell (FILE *stream);
```

Class

Standard C Function

Description

The ftell function returns the current value of the file position indicator for *stream*.

This particular implementation of ftell is simply a stub which always returns 0. On failure, ftell returns -1 and stores an appropriate error indicator in the static variable errno.

Notes

The function ftell is implemented simply as a stub which returns 0. If it is used, it will generate a link-time error indicating that the _WARNING_ftell_stub_used symbol is unresolved. If it is executed, it will write "WARNING - ftell (stub routine) called" to stderr. You must provide your own ftell function to allow it to work in your environment.

ftoa — Converts Floating-Point Number to ASCII String

Syntax

```
#include <mriext.h>
int ftoa (double value, char *str, char format, int prec);
```

Class

Microtec Research C Extension

Description

The ftoa function converts the floating-point number *value* to a NULL-terminated ASCII string with a specified format and precision. The output is placed in the area pointed to by *str*. No leading blanks or zeros are produced.

The conversion format character *format* (must be f, e, E, g, or G) and the precision *prec* are the same as those defined for the printf function. (The parameter *prec* is the same as the *digits* parameter in printf.) The printf function calls ftoa to convert floating-point numbers.

The ftoa function returns the number of characters placed in the output string, excluding the NULL terminator.

Notes

If an invalid floating-point number is passed to the ftoa function, the behavior is unpredictable.

fwrite — Writes to a Specified File

Syntax

```
#include <stdio.h>
size_t fwrite (const void *ptr, size_t size, size_t nitems, FILE *file);
```

Class

Standard C Function

Description

The fwrite function writes *nitems* (items of data) from the memory area pointed to by *ptr* to the file associated with *file*. The number of bytes in each item is specified by *size*.

The fwrite function returns the number of items written, which may be less than *nitems* if a write error is encountered.

Notes

The function fwrite can modify the static array _iob; no other static data is modified.

getc — Reads a Character from a File

Syntax

```
#include <stdio.h>
int getc (FILE *file);
```

Class

Standard C Function (implemented as a function and as a macro)

Description

The getc function or macro returns the next character (byte) from the named input *file*. It also moves the file pointer, if defined, ahead one character in *file*. If there is a problem with macro argument side effects, the fgetc function should be used instead.

The macro is invoked by default. If you want to call the function, you must either:

- enclose the name in parentheses:

```
(getc) (file);
```

or

- undefine the `getc` macro with `#UNDEF` before using it:

```
#UNDEF getc
```

When getc detects an error or reaches an End-Of-File, it returns EOF. If successful, getc returns the next character from the input file, or EOF if End-Of-File is detected.

Notes

The function getc can modify the static array _iob; no other static data is modified.

getchar — Reads a Character from Standard Input

Syntax

```
#include <stdio.h>
int getchar (void);
```

Class

Standard C Function (implemented as a function and as a macro)

Description

The getchar function or macro returns the next character from the standard input file, stdin. If there is a problem with macro argument side effects, the fgetc function should be used instead.

The macro is invoked by default. If you want to call the function, you must either:

- enclose the name in parentheses:

```
(getchar)();
```

or

- undefine the `getchar` macro with `#UNDEF` before using it:

```
#UNDEF getchar
```

When successful, getchar returns the next character byte from the terminal. When getchar fails or detects End-Of-File, it returns EOF.

Notes

The function getchar can modify the static array _iob; no other static data is modified.

getl — Reads a Long Integer from a Stream

Syntax

```
#include <mriext.h>
long getl (FILE * stream);
```

Class

Microtec Research C Extension (implemented as a function)

Description

The function getl reads a `long` integer from *stream* and returns it. getl returns EOF if the End-Of-File is read. Any `long` integer read from the stream must have been written with a putl function.

Notes

The function getl will yield unpredictable results if *stream* does not point to an open file.

The function getl can modify the static array _iob; no other static data is modified.

gets — Reads a Line from Standard Input

Syntax

```
#include <stdio.h>
char *gets (char *cp);
```

Class

Standard C Function

Description

The gets function reads characters from the standard file, stdin, into the buffer *cp* until End-Of-File is encountered or a new-line (\n) character is read. Any new-line character is discarded and a NULL character is appended so that *cp* contains a proper character string.

The gets function normally returns the pointer to *cp*.

When gets encounters a read error or End-Of-File, it returns a NULL pointer. If the area indicated by *cp* is not large enough to hold the result, gets will overwrite memory.

Notes

The function gets can modify the static array _iob; no other static data is modified.

getw — Reads a Short Integer from a Stream

Syntax

```
#include <mriext.h>
int getw (FILE *stream);
```

Class

Microtec Research C Extension

Description

The function getw reads a short int from *stream* and returns it as an `int`. getw returns EOF if the End-Of-File is read.

Any short integer read from *stream* must have been written with a putw function.

Notes

The function getw will yield unpredictable results if *stream* does not point to an open file.

The function getw can modify the static array _iob; no other static data is modified.

gmtime — Breaks Down Calendar Time to Local Time

Syntax

```
#include <time.h>
struct tm *gmtime (const time_t *timer);
```

Class

Standard C Function

Description

The gmtime function converts the calendar time pointed to by *timer* to a time expressed as Greenwich Mean Time (GMT). GMT is the time used by UNIX systems.

The gmtime function returns a pointer to a structure which contains the time. The calendar time pointed to by *timer* should be in seconds and should start at 00:00:00 (local time), January 1, 1970. The calendar time is normally set by a call to the time function.

The gmtime function is identical to the localtime function because time zones are target operating system-dependent and are not accounted for by this function.

Notes

The structure `tm` is in a static data area whose content is overwritten by each call. This function is similar but not fully compatible with the UNIX implementation. Time zones and Daylight Savings Time are target operating system-dependent and are not accounted for in this function.

The function gmtime modifies the static struct _xtm; no other static data is modified.

isalnum — Tests for an Alphanumeric Character

Syntax

```
#include <ctype.h>
int isalnum (int c);
```

Class

Standard C Function (implemented as a function and as a macro)

Description

The **isalnum** function or macro returns non-zero if the character in *c* is an alphanumeric character. It returns 0 otherwise. Alphanumeric characters are A through Z, a through z, and 0 through 9.

The macro is invoked by default. If you want to call the function, you must either:

- enclose the name in parentheses:

```
(isalnum)(c);
```

or

- undefine the `isalnum` macro with `#UNDEF` before using it:

```
#UNDEF isalnum
```

Notes

If *c* is outside the range -1 to 255, the result is undefined.

isalpha — Tests for an Upper- or Lower-Case Character

Syntax

```
#include <ctype.h>
int isalpha (int c);
```

Class

Standard C Function (implemented as a function and as a macro)

Description

The **isalpha** function returns non-zero if the character in *c* is an alphabetic character. It returns 0 otherwise. Alphabetic characters are A through Z and a through z.

The macro is invoked by default. If you want to call the function, you must either:

- enclose the name in parentheses:

```
(isalpha)(c);
```

or

- undefine the `isalpha` macro with `#UNDEF` before using it:

```
#UNDEF isalpha
```

Notes

If *c* is outside the range -1 to 255, the result is undefined.

isascii — Tests for an ASCII Character

Syntax

```
#include <mriext.h>
int isascii (int c);
```

Class

Microtec Research C Extension (implemented as a function and as a macro)

Description

The isascii function or macro returns non-zero if the character in *c* is an ASCII character. It returns 0 otherwise. ASCII characters range in value from 0 to 127.

The macro is invoked by default. If you want to call the function, you must either:

- enclose the name in parentheses:

```
(isascii)(c);
```

or

- undefine the `isascii` macro with `#UNDEF` before using it:

```
#UNDEF isascii
```

Notes

None.

iscntrl — Tests for a Control Character

Syntax

```
#include <ctype.h>
int iscntrl (int c);
```

Class

Standard C Function (implemented as a function and as a macro)

Description

The iscntrl function or macro returns non-zero if the character in *c* is a control character. It returns 0 otherwise. Control characters range in value from 0 to 31.

The macro is invoked by default. If you want to call the function, you must either:

- enclose the name in parentheses:

```
(iscntrl)(c);
```

or

- undefine the `iscntrl` macro with `#UNDEF` before using it:

```
#UNDEF iscntrl
```

Notes

If *c* is outside the range -1 to 255, the result is undefined.

isdigit — Tests for a Digit

Syntax

```
#include <ctype.h>
int isdigit (int c);
```

Class

Standard C Function (implemented as a function and as a macro)

Description

The **isdigit** function or macro returns non-zero if the character in *c* is a digit character. It returns 0 otherwise. Digit characters are **0** through **9**.

The macro is invoked by default. If you want to call the function, you must either:

- enclose the name in parentheses:

```
(isdigit)(c);
```

or

- undefine the `isdigit` macro with `#UNDEF` before using it:

```
#UNDEF isdigit
```

Notes

If *c* is outside the range -1 to 255, the result is undefined.

isgraph — Tests for a Printable Argument

Syntax

```
#include <ctype.h>
int isgraph (int c);
```

Class

Standard C Function (implemented as a function and as a macro)

Description

The **isgraph** function or macro returns non-zero if the character in *c* is a graphic character. It returns 0 otherwise. Printable graphic characters are ! through ~, and range in value from 33 through 126.

The macro is invoked by default. If you want to call the function, you must either:

- enclose the name in parentheses:

```
(isgraph)(c);
```

or

- undefine the `isgraph` macro with `#UNDEF` before using it:

```
#UNDEF isgraph
```

Notes

If *c* is outside the range -1 to 255, the result is undefined.

islower — Tests for a Lower-Case Character

Syntax

```
#include <ctype.h>
int islower (int c);
```

Class

Standard C Function (implemented as a function and as a macro)

Description

The islower function or macro returns non-zero if the character in *c* is a lower-case alphabetic character. It returns 0 otherwise. Lower-case characters are a through z.

The macro is invoked by default. If you want to call the function, you must either:

- enclose the name in parentheses:

```
(islower)(c);
```

or

- undefine the `islower` macro with `#UNDEF` before using it:

```
#UNDEF islower
```

Notes

If *c* is outside the range -1 to 255, the result is undefined.

isprint — Tests for a Printable Argument (Includes Blank Character)

Syntax

```
#include <ctype.h>
int isprint (int c);
```

Class

Standard C Function (implemented as a function and as a macro)

Description

The isprint function or macro returns non-zero if the character in *c* is a printable character. It returns 0 otherwise. Printable characters are " " (the blank character) through ~ and range in value from 32 to 126.

The macro is invoked by default. If you want to call the function, you must either:

- enclose the name in parentheses:

```
(isprint)(c);
```

or

- undefine the `isprint` macro with `#UNDEF` before using it:

```
#UNDEF isprint
```

Notes

If *c* is outside the range -1 to 255, the result is undefined.

ispunct — Tests for a Punctuation Mark

Syntax

```
#include <ctype.h>
int ispunct (int c);
```

Class

Standard C Function (implemented as a macro and as a function)

Description

The ispunct function or macro returns non-zero if the character in *c* is a punctuation character. It returns 0 otherwise. Punctuation characters are:

```
!  "  #  $  %  &  '  (  )  *  +  ,  -  .  /
:  ;  <  =  >  ?  @
[  \  ]  ^  _  `
{  |  }  ~
```

The macro is invoked by default. If you want to call the function, you must either:

- enclose the name in parentheses:

```
(ispunct)(c);
```

or

- undefine the `ispunct` macro with `#UNDEF` before using it:

```
#UNDEF ispunct
```

Notes

If *c* is outside the range -1 to 255, the result is undefined.

isspace — Tests for White Space

Syntax

```
#include <ctype.h>
int isspace (int c);
```

Class

Standard C Function (implemented as a function and as a macro)

Description

The isspace function or macro returns non-zero if the character in *c* is a space character. It returns 0 otherwise. Space characters are:

Character	Decimal Value
horizontal tab	9
new-line (line feed)	10
vertical tab	11
form-feed	12
carriage return	13
blank	32

The macro is invoked by default. If you want to call the function, you must either:

- enclose the name in parentheses:

```
(isspace)(c);
```

or

- undefine the `isspace` macro with `#UNDEF` before using it:

```
#UNDEF isspace
```

Notes

If *c* is outside the range -1 to 255, the result is undefined.

isupper — Tests for Upper-Case Character

Syntax

```
#include <ctype.h>
int isupper (int c);
```

Class

Standard C Function (implemented as a function and as a macro)

Description

The **isupper** function or macro returns non-zero if the character in *c* is an upper-case alphabetic character. It returns 0 otherwise. Upper-case alphabetic characters are A through Z.

The macro is invoked by default. If you want to call the function, you must either:

- enclose the name in parentheses:

```
(isupper)(c);
```

or

- undefine the `isupper` macro with `#UNDEF` before using it:

```
#UNDEF isupper
```

Notes

If *c* is outside the range -1 to 255, the result is undefined.

isxdigit — Tests for a Hexadecimal Digit

Syntax

```
#include <ctype.h>
int isxdigit (int c);
```

Class

Standard C Function (implemented as a function and as a macro)

Description

The isxdigit function or macro returns non-zero if the character in *c* is a hexadecimal digit character. It returns 0 otherwise. Hexadecimal digit characters are 0 through 9, A through F, and a through f.

The macro is invoked by default. If you want to call the function, you must either:

- enclose the name in parentheses:

```
(isxdigit)(c);
```

or

- undefine the `isxdigit` macro with `#UNDEF` before using it:

```
#UNDEF isxdigit
```

Notes

If *c* is outside the range -1 to 255, the result is undefined.

itoa — Converts Integer to ASCII String

Syntax

```
#include <mriext.h>
int itoa (int i, char *cp);
```

Class

Microtec Research C Extension

Description

The itoa function converts the signed integer *i* to a NULL-terminated ASCII string and places it in the area pointed to by *cp*. The pointer *cp* must point to a string.

The itoa function returns the number of characters placed in the output string, excluding the NULL terminator.

Notes

None.

itostr — Converts Unsigned Integer to ASCII String

Syntax

```
#include <mriext.h>
int itostr (unsigned u, char *cp, int base);
```

Class

Microtec Research C Extension

Description

The itostr function converts the `unsigned` integer *u* to a NULL-terminated ASCII string. The base number is specified by *base*, and the output is placed in the area pointed to by *cp*. The pointer *cp* must point to a string, and the specified base must be between 2 and 36 (otherwise, base 10 is assumed).

The itostr function returns the number of characters placed in the output string, excluding the NULL terminator.

Notes

None.

labs — Computes Absolute Value of a Long Integer

Syntax

```
#include <stdlib.h>
long int labs (long int j);
```

Class

Standard C Function

Description

The labs function returns the absolute value of *j*. If *j* is less than zero, abs(*j*) equals *-j*. If *j* is greater than or equal to zero, abs(*j*) equals *j*.

The labs function is similar to the abs function, except that the argument and the returned value each have the type `long int`.

Notes

The labs function returns an incorrect result when *j* is -2147483648.

ldexp — Computes Product of a Floating-Point Number and a Power of Two

Syntax

```
#include <math.h>
double ldexp (double d, int exp);
```

Class

Standard C Function

Description

The **ldexp** function calculates the value of $(d*(2^{exp}))$.

If an overflow results, the function returns positive or negative **HUGE_VAL** (the sign depends on the sign of the value). If an underflow results, the function returns 0. In both cases, **errno** is set to **ERANGE**.

Notes

The function **ldexp** can modify the static variable **errno**; no other static data is modified.

For information on the single-precision version of this function, **ldexpf**, refer to the *mathf.h* section at the beginning of this chapter.

ldiv — Computes Quotient and Remainder for a Division

Syntax

```
#include <stdlib.h>
ldiv_t ldiv (lont int numer, long int denom);
```

Class

Standard C Function

Description

The **ldiv** function computes the quotient and remainder of the division of the numerator *numer* by the denominator *denom*. If the division is inexact, the resulting quotient is the integer of lesser magnitude that is the nearest to the algebraic quotient. For example:

`quot * denom + rem = numer`

The **ldiv** function returns a structure of type `ldiv_t`. The structure contains the following members:

```
long int quot;
long int rem;
```

Notes

If the result cannot be represented, the behavior is undefined.

localtime — Breaks Down Calendar Time to Local Time

Syntax

```
#include <time.h>
struct tm *localtime (const time_t *timer);
```

Class

Standard C Function

Description

The **localtime** function converts the calendar time pointed to by *timer* to a time expressed as local time. A pointer to a structure which contains the local time is returned. The calendar time pointed to by *timer* should be in seconds and should start at 00:00:00 (local time), January 1, 1970. The calendar time is normally set by a call to the time function.

Notes

The structure `tm` is in a static data area whose contents are overwritten by each call. This function is similar but not fully compatible with the UNIX implementation. Time zones and Daylight Savings Time are target operating system-dependent and are not accounted for in this function.

The function **localtime** modifies the static struct _xtm; no other static data is modified.

log — Computes Natural Logarithm of a Number

Syntax

```
#include <math.h>
double log (double d);
```

Class

Standard C Function

Description

The log function returns the natural logarithm of *d*. The range of *d* is all positive numbers.

If the argument is negative, errno is set to EDOM. If the argument is zero, errno is set to ERANGE. The return value is zero for both cases.

Notes

The function log can modify the static variable errno; no other static data is modified.

For information on the single-precision version of this function, logf, refer to the *mathf.h* section at the beginning of this chapter.

log10 — Computes Base 10 Logarithm of a Number

Syntax

```
#include <math.h>
double log10 (double d);
```

Class

Standard C Function

Description

The **log10** function returns the base 10 logarithm of *d*. The range of *d* is all positive numbers.

If the argument is negative, **errno** is set to **EDOM**. If the argument is zero, **errno** is set to **ERANGE**. The return value is zero for both cases.

Notes

The function **log10** can modify the static variable **errno**; no other static data is modified.

For information on the single-precision version of this function, **log10f**, refer to the *mathf.h* section at the beginning of this chapter.

longjmp — Jumps Outside of a Function

Syntax

```
#include <setjmp.h>
void longjmp (jmp_buf env, int result);
```

Class

Standard C Function

Description

The **longjmp** function transfers control to *env* which specifies the location of a **setjmp** call, and the environment (e.g., visible variables, register contents) in effect at that call. Also, **longjmp** causes the value *result* to appear to be the result of the **setjmp** call. If *result* is zero, **longjmp** returns a 1.

Notes

Functions and operations that can cause errors, such as division by zero or the log of a negative number, do not have a convention for calling **longjmp**.

lseek — Sets the Current Location in a File

Syntax

```
long lseek (int fd, long offset, int whence);
```

Class

System Function

Description

The current position in the file is updated to the value given in *offset*. The *whence* argument indicates how to interpret the offset:

whence = 0	*offset* is relative to the beginning of the file
whence = 1	*offset* is relative to the current position
whence = 2	*offset* is relative to the end of the file

The lseek function returns the resulting file position as measured in bytes from the beginning of the file. If a failure occurs, -1 is returned.

Notes

The function lseek is implemented as a stub which returns 0. If it is used, it will generate a link-time error indicating that the _WARNING_lseek_stub_used symbol is unresolved. If it is executed, it will write "WARNING - lseek (stub routine) called" to stderr. You must provide your own lseek function to allow it to work in your environment.

The function lseek can modify the static array _iob; no other static data can be modified.

ltoa — Converts Long Integer to ASCII String

Syntax

```
#include <mriext.h>
int ltoa (long value, char *cp);
```

Class

Microtec Research C Extension

Description

The ltoa function converts the `long` signed integer *value* to a NULL-terminated ASCII string and places it in the area pointed to by *cp*. The pointer *cp* must point to a string.

The ltoa function returns the number of characters placed in the output string, excluding the NULL terminator.

Notes

None.

ltostr — Converts Unsigned Long Integer to ASCII String

Syntax

```
#include <mriext.h>
int ltostr (unsigned long ul, char *cp, int base);
```

Class

Microtec Research C Extension

Description

The ltostr function converts the `unsigned long` integer *ul* to a NULL-terminated ASCII string. The base number is specified by *base*, and the output is placed in the area pointed to by *cp*. The pointer *cp* must point to a string, and the specified base must be between 2 and 36 (otherwise, base 10 is assumed).

The ltostr function returns the number of characters placed in the output string, excluding the NULL terminator.

Notes

None.

malloc — Allocates Data Space Dynamically

Syntax

```
#include <stdlib.h>
void *malloc (size_t size);
```

Class

Standard C Function

Description

The malloc function allocates a new area of memory for program use and returns a pointer to that area. It attempts to allocate *size* bytes. If it is successful, it returns a pointer to that area. The space allocated is guaranteed to start on a boundary suitable for aligning any type.

If the requested amount of space cannot be allocated, malloc returns a NULL pointer. Allocations are provided from the heap data structure in the heap section.

Notes

The function malloc can modify the static struct _membase, or the static data _avail or _badlist; no other static data is modified.

max — Returns the Greater of Two Values

Syntax

```
#include <mriext.h>
max (a, b);
```

Class

Microtec Research C Extension (implemented as a macro)

Description

The macro max returns the larger of *a* and *b*. *a* and *b* can be any valid C expression yielding an integral, floating-point, or pointer type. max performs any type promotion necessary to make *a* and *b* the same type and returns that type. If either argument is a pointer type, both arguments must be pointers to the same type.

Notes

Both *a* and *b* are evaluated more than once.

memccpy — Copies Characters from Memory

Syntax

```
#include <mriext.h>
char *memccpy (char *s1, const char *s2, int c, size_t i);
```

Class

Microtec Research C Extension

Description

The memccpy function copies characters from memory area *s2* into *s1*, stopping after the first occurrence of character *c* has been copied, or after *i* characters have been copied, whichever comes first. This function returns a pointer to the character after the copy of *c* in *s1*.

The memccpy function returns a NULL pointer if *c* was not found in the first *i* characters of *s2*.

Notes

None.

memchr — Searches Character in Memory

Syntax

```
#include <string.h>
void *memchr (const void *cp, int c, size_t i);
```

Class

Standard C Function

Description

The **memchr** function locates the first occurrence of character *c* in the first *i* bytes of memory area *cp*. This function returns a pointer to the located character.

The **memchr** function returns a NULL pointer if the character does not occur in the first *i* bytes of memory area *cp*.

Notes

None.

memclr — Clears Memory Bytes

Syntax

```
#include <mriext.h>
char *memclr (char *cp, size_t i);
```

Class

Microtec Research C Extension

Description

The memclr function sets the first *i* bytes in memory area *cp* to zero. The memclr function returns *cp*.

Notes

None.

memcmp — Compares Characters in Memory

Syntax

```
#include <string.h>
int memcmp (const void *s1, const void *s2, size_t i);
```

Class

Standard C Function

Description

The **memcmp** function compares its arguments, looking at the first *i* characters only.

The **memcmp** function returns an integer less than, equal to, or greater than 0 if *s1* is lexicographically less than, equal to, or greater than *s2*.

Notes

None.

memcpy — Copies Characters from Memory

Syntax

```
#include <string.h>
void *memcpy (void *s1, void *s2, size_t i);
```

Class

Microtec Research C Extension

Description

The memcpy function copies *i* characters from memory area *s2* to *s1*. The memcpy function returns *s1*.

Notes

If copying takes place between memory locations that overlap, the behavior is undefined.

memmove — Copies Characters from Memory

Syntax

```
#include <string.h>
void *memmove (void *s1, const void *s2, size_t n);
```

Class

Standard C Function

Description

The function memmove copies *n* characters from the object pointed to by *s2* to the object pointed to by *s1*. The copying occurs as though the data in the object pointed to by *s2* were first copied into a temporary array of *n* characters and then copied from that array into the object pointed to by *s1*.

The memmove function returns the value of *s1*.

Notes

None.

memset — Sets the Value of Characters in Memory

Syntax

```
#include <string.h>
void *memset (void *cp, int c, size_t i);
```

Class

Standard C Function

Description

The memset function sets the first *i* characters in memory area *cp* to the value of character *c*. The memset function returns *cp*.

Notes

None.

min — Returns the Lesser of Two Values

Syntax

```
#include <mriext.h>
min (a, b);
```

Class

Microtec Research C Extension (implemented as a macro)

Description

The macro **min** returns the lesser of *a* and *b*. *a* and *b* can be any valid C expression. *a* and *b* can be any integral, floating-point, or pointer type. **min** performs any type promotion necessary to make *a* and *b* the same type and returns that type. If either argument is a pointer type, both arguments must be pointers to the same type.

Notes

Both *a* and *b* are evaluated more than once.

modf — Breaks Down a Number into Fractional and Integer Parts

Syntax

```
#include <math.h>
double modf (double d, double *dp);
```

Class

Standard C Function

Description

The **modf** function returns the signed fractional portion of *d*, and breaks down *d* into fractional and integer parts. The signed fractional portion of *d* is returned. The integer portion is stored as a floating-point value at *dp*.

There is no error return.

Notes

For information on the single-precision version of this function, **modff**, refer to the *mathf.h* section at the beginning of this chapter.

offsetof — Determines Offset of Structure Member

Syntax

```
#include <stddef.h>
size_t offsetof (type, member);
```

Class

Standard C Macro (implemented as a macro only)

Description

The offsetof function yields the offset (in bytes) to the structure member *member* from the beginning of the structure. For example, given the declaration:

```
typedef struct {int a; double b; int c} STRUCT;
```

the following expression evaluates to 12 on a 32-bit target and 10 on a 16-bit target:

```
offsetof (STRUCT,c)
```

Notes

None.

open — Opens a Specified File

Syntax

```
int open (char *filename, int mode);
```

Class

System Function

Description

The open function opens an existing file for reading, writing, or updating. The NULL-terminated string *filename* must correspond to a valid file name on the target operating system.

The function opens the file named by *filename* in the mode indicated by *mode*. The values for *mode* and the corresponding symbols are:

Value	Symbol	Function
0x0000	O_RDONLY	Opens for read only
0x0001	O_WRONLY	Opens for write only
0x0002	O_RDWR	Opens for read and write
0x0008	O_APPEND	Performs writes at EOF
0x0200	O_CREAT	Creates a file
0x0400	O_TRUNC	Truncates the file
0x4000	O_FORM	Is a text file
0x8000	O_BINARY	Is a binary file

The open function is a stub and always returns 0. It should return -1 if the file does not exist, if the file cannot be read or written, or if too many files are open.

Notes

Since embedded systems do not usually have a file system, the open function is provided as a stub only; it does nothing and always returns a 0. If it is used, it will generate a link-time error indicating that the _WARNING_open_stub_used symbol is unresolved. If it is executed, it will write "WARNING - open (stub routine) called" to stderr. You must provide your own open function to allow it to work in your environment.

If you are developing code for an environment that does have a file system, you can provide a replacement for the open function stub by linking in your implementation of open before the run-time library.

The function open can modify the static array _iob; no other static data is modified.

perror — Writes Error Message to Standard Error Device

Syntax

```
#include <stdio.h>
void perror (const char *S);
```

Class

Standard C Function

Description

The perror function writes an error message to the standard error device. The contents of the error message strings are the same as those returned by the strerror function with argument errno.

If the argument *s* is not a null pointer, perror first writes the string pointed to by *s* followed by a colon (:) and a space. Then, whether or not *s* is a null pointer, perror writes the appropriate error message string followed by a new-line character.

The perror function returns no value.

Notes

None.

pow — Computes d1 Raised to the Power d2

Syntax

```
#include <math.h>
double pow (double d1, double d2);
```

Class

Standard C Function

Description

The pow function returns the result of raising *d1* to the power *d2*.

Infinity is returned and a domain error occurs when *d1* is zero and *d2* is less than zero. NaN is returned and a domain error occurs when *d1* is negative and *d2* is not an integer. A range error can occur if the result is too large or too small.

Notes

The function pow can modify the static variable errno; no other static data is modified.

For information on the single-precision version of this function, powf, refer to the *mathf.h* section at the beginning of this chapter.

printf — Formatted Print on Standard Output

Syntax

```
#include <stdio.h>
int printf (const char format,...);
```

Class

Standard C Function

Description

The **printf** function provides formatted output to the standard output file. It simplifies type conversions, scaling, and the positioning of output within ASCII character records.

The **printf** function requires a character string called a *format* as the first parameter. This format contains plain characters and conversion specifications. **printf** begins at the left of the format and copies plain characters to the output file. Whenever it encounters a conversion specification, it takes the next unused parameter, converts it according to the specification, and copies the result to the output file.

The conversion specifiers have the following format:

% [*flags*] [*digits1*] [. [*digits2*]] [l] *conversion_char*

where:

%	Signals the start of a conversion specification.
flags	Refers to zero or more *flags* (in any order) that modify the meaning of the conversion specification. The *flags* and their meanings are described in Table 5-26.
digits1	Represents a string of one or more ASCII decimal characters. The first *digits* is the minimum field width. The field will be at least this many characters wide, padded if necessary. The padding is on the left unless the - character is specified. If - is specified, padding is on the right. The field is padded with blanks unless the first digit in *digits* is a 0. If it is a 0, the field is padded with zeros.
.	Separates the two digit strings and must be specified if the second digit string is specified.

Table 5-26. Conversion Specification Flags

Flag	Meaning
-	The conversion will be left-justified within the field. It will be right-justified if this flag is not specified.
+	The result of a signed conversion will always begin with a plus or minus sign. If this flag is not specified, it will begin with a sign only when a negative value is converted.
"white space"	If the first character of a signed conversion is not a sign, or if a signed conversion results in no characters, a white space will be prefixed to the result. If the "white space" and + flags both appear, the white space flag will be ignored.
#	The result will be converted to an "alternate form". For o conversion, it increases the precision to force the first digit of the result to be a zero. For the x (or X) conversion, a non-zero result will have a 0x (or 0X) prefixed to it. For the e, E, f, g, and G conversions, the result will always contain a decimal point character, even if no digits follow it (normally, a decimal point character appears in the result of these conversions only if a digit follows it). For the g and G conversions, trailing zeros will not be removed from the result. For other conversions, the behavior is undefined.
0	For the d, i, o, u, x, X, e, E, f, g, and G conversions, leading zeros (following any indicator sign or base) are used to pad to the field width; no space padding is performed. If the 0 and - flags both appear, the 0 flag will be ignored. For the d, i, o, u, x, and X conversions, if a precision is specified, the 0 flag will be ignored. For other conversions, the behavior is undefined.

digits2 Specifies the minimum number of digits to appear for the d, o, u, x, and X conversions. For the e, E, and f conversions, it is the number of digits to appear after the decimal point. For the g and G conversions, it is the maximum number of significant digits. For strings, it is the maximum number of characters to be printed.

l (letter ell) Indicates that the following conversion character (d, o, x, X, or u) corresponds to a long integer argument.

conversion_char Indicates the characters shown in Table 5-27.

Table 5-27. Conversion Characters for printf

Character	Meaning
%	Substitutes the character % for the field. In addition, any other non-conversion character following a % is printed. %% is used to get a % in the output as a literal character.
c	Interprets argument as a single character.
d, i	Converts argument to decimal format.
e, E	Interprets argument as **double** and converts to e or E notation (see below).
f	Interprets argument as **double** and converts to f notation (see below).
g, G	Interprets argument as **double** and prints in e or f notation (or E notation in the case of G format code), whichever is shorter. Non-significant zeros are not printed. A decimal point is printed only if it is followed by a digit.
n	Interprets argument as a pointer to an integer; this integer contains the number of characters written to the output stream so far by this call to **printf**.
o	Converts argument to octal format (with the leading zero).
p	Interprets argument as a pointer to **void**. The value of the pointer is printed.
s	Interprets argument as a string. The characters from the string are copied to the output until a **NULL** character is reached or the maximum number of characters specified have been printed.
u	Converts argument to unsigned decimal format.
x	Converts argument to hexadecimal format, using decimal digits and the letters a, b, c, d, e, or f.
X	Converts argument to hexadecimal format, using decimal digits and the letters A, B, C, D, E, or F.

Values printed in e or E notation have the following format:

[-] *d . d.... e* {+ | -} *dd*

Each *d* represents a decimal digit. The number is first scaled to give one digit to the left of the decimal point. The number of digits in the fractional part is specified by the precision, which is six by default, or the minimum field width if specified. If the precision is zero, no decimal point character is printed. The exponent always has at least two digits and is zero-filled if necessary. The sign for the mantissa is printed only if the number as a whole is negative. The sign for the exponent is always printed. The E format code will produce a number with E instead of e introducing the exponent.

Values printed in f notation have the following format:

[-] *d.... . d....*

Each *d* represents a decimal digit. The number of digits in the fractional part is specified by the precision, which is six by default, or the minimum field width if specified. If the precision is explicitly zero, no decimal point character is printed. If a decimal point character is printed, at least one digit appears before it.

Floating-point values are converted to double precision because they are being passed as parameters. Other values can also be converted.

When **printf** is successful, it returns the number of characters transmitted. If an output error is encountered, **printf** returns a negative number.

Notes

Since **printf** uses the format to decide how many arguments to print, the number of conversion specifications must equal the number of remaining arguments. If there are too many arguments, some of them will not get printed. If there are too few, other parts of the stack will be used as arguments for output. In the case of a %s conversion, this can result in a very long string.

If the type of an argument does not correspond to its field specification, an incorrect conversion will occur.

The **printf** function fails if a field is wider than 512 characters.

If an invalid floating-point number is passed to the **printf** function, the behavior is undefined.

The function **printf** can modify the static array _iob; no other static data is modified.

putc — Writes a Character to a File

Syntax

```
#include <stdio.h>
int putc (int c, FILE *file);
```

Class

Standard C Function (implemented as a function and as a macro)

Description

The **putc** function or macro writes one character to the file indicated by *file*. The character *c* is written to the file and returned as the normal value of **putc**. If there is a problem with macro argument side effects, the **fputc** function can be used instead.

The macro is invoked by default. If you want to call the function, you must either:

- enclose the name in parentheses:

```
(putc) (c,file);
```

or

- undefine the `putc` macro with `#UNDEF` before using it:

```
#UNDEF putc
```

The **putc** function fails if the file cannot be written for any reason. It returns EOF (End-Of-File) when it fails; otherwise, it returns the character written.

Notes

The function **putc** can modify the static array _iob; no other static data is modified.

putchar — Writes a Character to Standard Output

Syntax

```
#include <stdio.h>
int putchar (int C);
```

Class

Standard C Function (implemented as a function and as a macro)

Description

The putchar function or macro writes one character to the standard output and returns that character. If there is a problem with macro argument side effects, the fputc function should be used instead.

The macro is invoked by default. If you want to call the function, you must either:

- enclose the name in parentheses:

```
(putchar)(C);
```

or

- undefine the `putchar` macro with `#UNDEF` before using it:

```
#UNDEF putchar
```

The putchar function fails if the file cannot be written for any reason. It returns an EOF (End-Of-File) when it fails; otherwise, it returns the character written.

Notes

The function putchar can modify the static array _iob; no other static data is modified.

putl — Writes a Long Integer to a Stream

Syntax

```
#include <mriext.h>
long putl (long l, FILE *stream);
```

Class

Microtec Research C Extension

Description

The function **putl** writes a `long` integer to *stream*. **putl** returns *l*.

Notes

The function **putl** will yield unpredictable results if *stream* does not point to an open file.

The function **putl** can modify the static array _iob; no other static data is modified.

puts — Writes a String to Standard Output

Syntax

```
#include <stdio.h>
int puts (const char *cp);
```

Class

Standard C Function

Description

The puts function writes a string to the standard output. It prints characters from the buffer *cp* to the standard output **stdout**, terminating on a NULL character, and replacing it with a line feed (newline).

The puts function fails if it cannot write the file for any reason. It returns EOF (End-Of-File) on error; otherwise, it returns zero.

Notes

The puts function can modify the static array **_iob**; no other static data is modified.

putw — Writes a Short Integer to a Stream

Syntax

```
#include <mriext.h>
int putw (int w, FILE *stream);
```

Class

Microtec Research C Extension

Description

The function **putw** writes a short integer to *stream*. **putw** returns *w*.

Notes

The function **putw** will yield unpredictable results if *stream* does not point to an open file.

The function **putw** can modify the static array **_iob**; no other static data is modified.

qsort — Performs Quick Sort Algorithm

Syntax

```
#include <stdlib.h>
void qsort (void *base, size_t nel, size_t size,
            int (*compar) (const void *, const void *));
```

Class

Standard C Function

Description

The qsort function is an implementation of the quick-sort algorithm. It sorts a table of data in place.

The element at the base of the table is pointed to by *base*. The number of elements in the table is specified by *nel*. The size of each table element is *size*.

You must create a comparison function (*compar*), which is called with two arguments that point to the elements being compared. Your *compar* function must return an integer less than, equal to, or greater than zero, indicating that the first argument is considered to be respectively less than, equal to, or greater than the second argument.

Notes

None.

rand — Pseudo-Random Number Generator

Syntax

```
#include <stdlib.h>
int rand (void);
```

Class

Standard C Function

Description

The **rand** function uses a multiplicative congruent pseudo-random number generator with a period of 2^{32}. The **rand** function returns the next pseudo-random number in a range from 0 to 32767.

Notes

The function **rand** modifies the static variable **_randx**; no other static data is modified.

read — Reads Bytes from a Specified File

Syntax

```
int read (int fd, char *buffer, int nbyte);
```

Class

System Function

Description

The read function reads *nbyte* bytes from the file associated with *fd* into the buffer pointed to by *buffer*. The file descriptor *fd* is returned by open or creat. Up to 64K bytes may be read, starting at the current file position.

The read function returns the number of bytes actually read, or 0 when End-Of-File is reached. On error, the read function returns -1.

Notes

Since embedded systems do not usually have a file system, the read function provided in the run-time library ignores the file descriptor *fd*, and reads from the variable _simulated_input. The source for the read function is provided on the distribution. If you want to modify the read function or replace it with one of your own, you should link in your implementation of read before the run-time library.

realloc — Changes the Size of Dynamically Allocated Storage

Syntax

```
#include <stdlib.h>
void *realloc (void *cp, size_t elsize);
```

Class

Standard C Function

Description

The realloc function changes the size of the object pointed to by *cp* to the size in bytes specified by *elsize*. The pointer *cp* must be a pointer returned previously by one of the allocation routines (see calloc, malloc, or zalloc).

The realloc function returns a pointer to the start (lowest byte address) of the possibly moved object. The pointer returned can safely be cast to point to an object of any type.

If *cp* is a null pointer, the realloc function behaves like the malloc function for the specified size. If *elsize* is zero and *cp* is not a null pointer, the object it points to is freed.

If the space cannot be allocated, realloc returns a NULL pointer and the object pointed to by *cp* is unchanged.

Notes

If *cp* does not match a pointer returned earlier by the calloc, malloc, realloc, or zalloc function, the behavior is undefined.

The function realloc can modify the static struct _membase, or static data _avail or _badlist; no other static data is modified.

remove — Makes File Name Inaccessible

Syntax

```
#include <stdio.h>
int remove (const char *filename);
```

Class

Standard C Function

Description

The remove function causes the file whose name is *filename* to no longer be accessible by that name. A subsequent attempt to open that file using that name will fail, unless it is created anew. If no other file names address the file in question, the file will be deleted.

The remove function returns zero if the operation succeeds; it returns non-zero if it fails.

Notes

The function remove calls the system function **unlink**, which is implemented as a stub. You must provide your own **unlink** function to allow remove to work in your environment.

rewind — Points to Beginning of File

Syntax

```
#include <stdio.h>
void rewind (FILE *stream);
```

Class

Standard C Function

Description

The rewind function sets the file pointed to by *stream* to the beginning of the file.

The rewind function returns no value.

Notes

The rewind function uses the lseek function which is implemented as a stub. You must provide your own lseek function to allow rewind to work in your environment.

The function rewind can modify the static array _iob; no other static data is modified.

sbrk — Allocates Memory Space

Syntax

```
void *sbrk (int size);
```

Class

System Function

Description

The sbrk function allocates additional memory from the system. A new block of memory, at least *size* bytes, is allocated and a pointer to the start of this memory block is returned.

The sbrk function returns a pointer to the starting address of the memory block, or -1 if the requested amount of memory cannot be allocated.

scanf — Provides Formatted Read from Standard Input

Syntax

```
#include <stdio.h>
int scanf (const char *format,...);
```

Class

Standard C Function

Description

The scanf function separates input text into fields, then converts those fields to their internal type.

The scanf function reads characters from the standard input, converting them as indicated by the format and storing the result in the place indicated by the remaining arguments.

> **Note**
>
> All arguments to scanf must be pointers. Generally, if an argument in the call is anything but a character string or other array reference, it must be preceded by the & operator to get its address.

The format for scanf is similar to the format for printf. It contains plain characters, white space, and conversion characters. It starts at the left of the format and matches plain characters to those in the input. Plain characters that are matched are otherwise ignored. Whenever scanf encounters a conversion specification, it takes the next input field, converts it according to the specification, and stores the result in the area indicated by the next unused pointer parameter. If the assignment suppression operator (*) is used, the field is read, but the value is not stored.

White space in the format is ignored, and any white space in the input delimits fields.

Conversion characters are listed in Table 5-28.

Table 5-28. Conversion Characters for scanf

Character	Meaning
%	Signals the start of a conversion specification. Any non-conversion character following a % is ignored. %% is used to match % in the input.
[	Interprets argument as a string composed of the characters between the opening bracket ([) and the closing bracket (]). If the character after the opening bracket is a circumflex (^), the string is interpreted as being composed of the characters not between the brackets. If a closing bracket immediately follows an opening bracket, it is interpreted as part of the string. If you want to include a circumflex in your string, it cannot be the first character after the opening bracket.
c	Interprets argument as a single character; no white space is skipped. To skip white space, use %1s.
d	Converts argument from decimal format.
E, e, f, G, g	Interprets argument as float or double and converts it from floating-point notation (see below).
i	Interprets argument as base 10 unless the sequence begins with 0x or 0X (implies base 16) or 0 (implies base 8).
n	Writes the number of characters read so far to the integer pointed to by the corresponding argument. The assignment count is not incremented.
o	Converts argument from octal format, and the leading zeros are ignored.
p	Interprets argument as a pointer to void.
s	Interprets argument as a string. The characters from the string are copied from the output until white space is reached or the maximum number of characters specified have been read. A NULL character is appended to the end of the string before storage.
u	Interprets argument as an unsigned decimal integer.
X, x	Converts argument from hexadecimal format, without the leading 0x.

The conversion specifiers have the following format:

% [*] [*digits*] [*size*] *conversion_char*

where:

%	Signals the start of a conversion specification.
*	Indicates assignment suppression.
digits	Represents a string of one or more ASCII decimal characters representing a maximum field width. For float and double precision numbers, it is the maximum number of decimal places to read; for non-floating types, it is the maximum number of characters to read.
size	Is a size character, which can be either l or h. If the size character is l (letter ell), the following d, o, or x conversion character corresponds to a long argument, and the following e or g conversion character corresponds to a double argument. If the size character is h, the following d, o, or x corresponds to a short int.

The floating-point notations for input are the same as the E and f notations for output, except that the sign for the mantissa can be either a + or - sign and that the %e, %f, and %g conversions can take input in either E or f notation.

Note

If the field specified is %le, %lf, or %lg, the corresponding argument should be a pointer to double precision.

The scanf function returns the number of successfully matched and assigned input items. If the assignment suppression operator is used on a field, it is not counted in the return value.

If no fields are successfully converted and assigned, scanf returns a 0. If the end of the input file is encountered, it returns EOF.

Notes

The function scanf can modify the static array _iob; no other static data is modified.

setbuf — Associates a Buffer with an I/O File

Syntax

```
#include <stdio.h>
void setbuf (FILE *file, char *cp);
```

Class

Standard C Function

Description

The setbuf function allows you to specify buffered I/O by associating a buffer with an input or output file. It can be used after the file has been opened, but must be used before any I/O occurs. It causes file operations to use the specified character array as a buffer instead of an automatically allocated buffer. The buffer must be large enough to hold an entire input record.

The BUFSIZ constant defined in the standard include file **stdio.h** is used as the size of the buffer. If *cp* is NULLPTR, the file will be unbuffered. Otherwise the buffer is used for all subsequent I/O operations on the file. A buffer is normally obtained by calling malloc.

Notes

The function setbuf can modify the static array _iob; no other static data is modified.

setjmp — Establishes a Label for longjmp

Syntax

```
include <setjmp.h>
int setjmp (jmp_buf env);
```

Class

Standard C Function

Description

The **setjmp** function saves the current environment (program counter, stack pointer, etc.) in *env*, and returns the value zero. When control returns to the point after the **setjmp** call due to a call to **longjmp**, the value of **setjmp** is *result*, as specified in the **longjmp** call.

Notes

Do not use **setjmp** as a parameter to a function call, or as an operand of any C expression that requires temporary values to be pushed. In such a case, there can be stack values that will not be restored when **longjmp** returns to this point.

Safe

```
i = setjmp(env);
switch (setjmp(env));
```

Unsafe

```
i = f(setjmp(env), i, j, k);
*p = setjmp(env); /* target address is pushed */
```

setlocale — Sets or Queries Local Information

Syntax

```
#include <locale.h>
char *setlocale (int category, const char *locale);
```

Class

Standard C Function

Description

The setlocale function allows the locale information to be changed or queried. The *locale* parameter is a pointer to a string which describes the locale which is to be set. Currently, the parameter *category* that indicates the portion of the program's locale to be changed or queried is ignored by the setlocale function. Normally, the following values for *category* are valid:

LC_ALL	Names the program's entire locale
LC_COLLATE	Affects strcoll and, if available, the strxfrm function
LC_TYPE	Affects character handling functions and multibyte functions
LC_MONETARY	Affects monetary formatting information
LC_NUMERIC	Affects decimal point character and non-monetary formatting information
LC_TIME	Affects the behavior of time functions

Currently, only the value of "C", " " (blank), and NULL are accepted for *locale*. "C" specifies the minimal environment for C, and a value of " " (blank) specifies the native environment. NULL indicates a query. If none of these three are specified, setlocale returns a NULL pointer.

If a NULL pointer is passed for *locale*, setlocale returns a pointer to a string associated with the specified *category* for the new locale. A subsequent call to setlocale with that string value as its associated category will restore that part of the program's locale. The string may not be modified by the program; however, it may be overwritten by a subsequent call to setlocale.

Notes

If the string pointed to by the return value of setlocale is modified, subsequent behavior is undefined.

This implementation of setlocale is the minimum required by ANSI C. This particular implementation has no effect on the libraries. It is provided for your convenience.

sin — Computes Sine of a Number

Syntax

```
#include <math.h>
double sin (double d);
```

Class

Standard C Function

Description

The sin function returns the sine of *d*. Arguments must be in radians.

Notes

A large magnitude argument can yield a result with little or no significance.

For information on the single-precision version of this function, sinf, refer to the *mathf.h* section at the beginning of this chapter.

sinh — Computes Hyperbolic Sine of a Number

Syntax

```
#include <math.h>
double sinh (double d);
```

Class

Standard C Function

Description

The **sinh** function returns the hyperbolic sine of *d*.

The **sinh** function returns HUGE_VAL when the correct value exceeds the physical limitations of the machine, and **errno** then is set to ERANGE.

Notes

The function **sinh** can modify the static variable **errno**; no other static data is modified.

For information on the single-precision version of this function, **sinhf**, refer to the *mathf.h* section at the beginning of this chapter.

sprintf — Formats a String

Syntax

```
#include <stdio.h>
int sprintf (char *cp, const char *format,...);
```

Class

Standard C Function

Description

The **sprintf** function does the formatting of **printf** while retaining the output in main memory. The parameter *cp* receives the formatted string followed by the NULL character. The results and the parameter *format* are the same as for **printf**.

When **sprintf** is successful, it returns the number of characters transmitted (not including the NULL character), or a -1 if it failed.

Notes

If the area indicated by *cp* is not large enough to hold the result, **sprintf** overwrites memory.

sqrt — Computes Square Root of a Number

Syntax

```
#include <math.h>
double sqrt (double d);
```

Class

Standard C Function

Description

The sqrt function returns the square root of *d*. The range of *d* is zero and all positive numbers.

If the argument is negative, errno is set to ERANGE and the function returns zero.

Notes

The function sqrt can modify the static variable errno; no other static data is modified.

For information on the single-precision version of this function, sqrtf, refer to the *mathf.h* section at the beginning of this chapter.

srand — Sets the Seed of the Random Number Generator

Syntax

```
#include <stdlib.h>
void srand (unsigned seed);
```

Class

Standard C Function

Description

The srand function sets the multiplicative congruent pseudo-random number generator to a random starting point *seed*.

Notes

The function srand modifies the static variable _randx; no other static data is modified.

sscanf — Provides Formatted String Conversion

Syntax

```
include <stdio.h>
int sscanf (const char *cp, const char *format, ...);
```

Class

Standard C Function

Description

The sscanf function does the parsing and conversion of scanf on data that is already in main memory. The parameter *cp* must contain a formatted string to be parsed by sscanf. The *format* and results are the same as for scanf.

The sscanf function returns the value of the macro EOF if an input failure occurs before any conversion. Otherwise, the sscanf function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early conflict between an input character and the format.

Notes

None.

strcat — Concatenates Two Strings

Syntax

```
#include <string.h>
char *strcat (char *s1, const char *s2);
```

Class

Standard C Function

Description

The strcat function appends a copy of the string *s2* to the end of the string *s1*. The initial character of *s2* overwrites the NULL character at the end of *s1*. A pointer to *s1* is returned.

Notes

No checking is done on the size of *s1*.

strchr — Locates First Occurrence of a Character in a String

Syntax

```
#include <string.h>
char *strchr (const char *S, int C);
```

Class

Standard C Function

Description

The strchr function returns a pointer to the first occurrence of the character *c* in the string *s*, if *c* occurs in *s*. The terminating NULL character is considered to be a part of the string.

If the character *c* does not occur in *s*, strchr returns a NULL pointer.

Notes

None.

strcmp — Compares Two Strings

Syntax

```
#include <string.h>
int strcmp (const char *s1, const char *s2);
```

Class

Standard C Function

Description

The strcmp function compares the string *s1* to the string *s2*. The strcmp function returns an integer less than, equal to, or greater than 0 if *s1* is lexicographically less than, equal to, or greater than *s2*. No check is made to see that the strings are NULL terminated.

Notes

None.

strcoll — Compares Two Strings

Syntax

```
#include <string.h>
int strcoll (const char *s1, const char *s2);
```

Class

Standard C Function

Description

The strcoll function compares the string *s1* to the string *s2*. The strcoll function returns an integer less than, equal to, or greater than 0 if *s1* is lexicographically less than, equal to, or greater than *s2*, respectively.

The strcoll function uses the collating sequence specified by the current locale. However, because only the "C" and "" (native) locales are supported, strcoll works exactly like strcmp.

Notes

None.

strcpy — Copies a String to Another String

Syntax

```
#include <string.h>
char *strcpy (char *s1, const char *s2);
```

Class

Standard C Function

Description

The strcpy function copies string *s2* to string *s1* until the NULL character is copied (copying from the right parameter to the left resembles an assignment statement).

A pointer to *s1* is returned.

Notes

No checking is done on the size of *s1*. If copying takes place between strings that overlap, the behavior is undefined.

strcspn — Computes Initial Segment Length of String

Syntax

```
#include <string.h>
size_t strcspn (const char *s1, const char *s2);
```

Class

Standard C Function

Description

The strcspn function computes the length of the initial segment of the string pointed to by *s1*, which consists entirely of characters not from the string pointed to by *s2*. The terminating NULL character is not considered to be a part of *s2*. The strcspn function returns the length of the segment.

Notes

None.

strerror — Points to Error Message

Syntax

```
#include <string.h>
char *strerror (int errnum);
```

Class

Standard C Function

Description

The strerror function returns a pointer to a string containing an error message. The value *errnum* is the error number. Error message strings correspond to error conditions which might be set into errno.

Notes

The string pointed to by strerror must not be modified by the program.

strlen — Returns String Length

Syntax

```
#include <string.h>
size_t strlen (const char *s);
```

Class

Standard C Function

Description

The strlen function returns the length of a string. It returns the number of characters in *s*, not including the terminating NULL character.

Notes

If the string is not properly terminated by a NULL character, the length returned is invalid.

strncat — Performs Limited Concatenation of Two Strings

Syntax

```
#include <string.h>
char *strncat (char *s1, const char *s2, size_t n);
```

Class

Standard C Function

Description

The strncat function appends a copy of not more than *n* characters from string *s2* to the end of string *s1*. A pointer to *s1* is returned.

The append ends when strncat encounters a NULL character or when *n* characters have been appended. A terminating NULL character is always appended to the result.

Notes

No checking is done on the size of *s1*.

strncmp — Performs Limited Comparison of Two Strings

Syntax

```
#include <string.h>
int strncmp (const char *s1, const char *s2, size_t n);
```

Class

Standard C Function

Description

The strncmp function returns an integer less than, equal to, or greater than 0 if *s1* is lexicographically less than, equal to, or greater than *s2*. No check is made to see that the strings are NULL terminated.

The strncmp function does not compare more than *n* characters from string *s1* to the string *s2*.

Notes

None.

strncpy — Performs Limited Copy of a String

Syntax

```
#include <string.h>
char *strncpy (char *s1, const char *s2, size_t n);
```

Class

Standard C Function

Description

The strncpy function copies string *s2* to string *s1* until the NULL character is copied, or until *n* characters are copied. If string *s2* is shorter than *n* characters, additional NULL characters are appended until *n* characters have been written.

A pointer to *s1* is returned.

Notes

No checking is done on the size of *s1*. If string *s2* is *n* characters or longer, the result will not be NULL terminated. If copying takes place between strings that overlap, the behavior is undefined.

strpbrk — Finds Break Character in String

Syntax

```
#include <string.h>
char *strpbrk (const char *s1, const char *s2);
```

Class

Standard C Function

Description

The **strpbrk** function returns a pointer to the first occurrence in string *s1* of any character from the string *s2*. Both strings must be NULL terminated.

If characters from *s2* do not exist in *s1*, the **strpbrk** function returns a NULL pointer.

Notes

None.

strrchr — Locates Last Occurrence of a Character in a String

Syntax

```
#include <string.h>
char *strrchr (const char *s, int c);
```

Class

Standard C Function

Description

The **strrchr** function returns a pointer to the last occurrence of character *c* in string *s*, if *c* occurs in *s*. The NULL-terminating character is considered to be a part of the string.

If the character *c* does not occur in *s*, **strrchr** returns a NULL pointer.

Notes

None.

strspn — Computes Initial Segment Length of String

Syntax

```
#include <string.h>
size_t strspn (const char *s1, const char *s2);
```

Class

Standard C Function

Description

The strspn function computes the length of the initial segment of the string pointed to by *s1*, which consists entirely of characters from the string pointed to by *s2*. The strspn function returns the length of the segment.

Notes

None.

strstr — Finds First Occurrence in String

Syntax

```
#include <string.h>
char *strstr (const char *s1, const char *s2);
```

Class

Standard C Function

Description

The strstr function locates the first occurrence in the string pointed to by *s1* of the sequence of characters, not including the terminating null character, in the string pointed to by *s2*.

The strstr function returns a pointer to the located string, or NULL if the string is not found. If *s2* points to a string with zero length, the function returns *s1*.

Notes

None.

strtod — Converts ASCII String to Double

Syntax

```
#include <stdlib.h>
double strtod (const char *nptr, char **endptr);
```

Class

Standard C Function

Description

The strtod function returns the double precision floating-point value of *nptr*. The pointer *nptr* must point to a string. The strtod function takes ASCII decimal digits from the left. The format of the string is:

[*ws*] [*sign*] *digits* ["." *digits*] [*exp* [*sign*] *digits*]

where:

ws	is one or more blanks or tabs.
sign	is a - or +.
digits	is one or more decimal digits.
exp	is e or E.

Any character other than those specified, in particular a NULL character, terminates the resulting number.

The address of the first character which does not conform to the above format is returned in the object pointed to by *endptr*, provided that *endptr* is not a NULL pointer.

If no conversion could be performed, 0 is returned and the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer. If an overflow occurred during the conversion, +/- HUGE_VAL is returned and errno is set to ERANGE. If an underflow occurs, 0.0 is returned and errno is set to ERANGE.

Notes

The function strtod can modify the static variable errno; no other static data is modified.

strtok — Locates Tokens in a String

Syntax

```
#include <string.h>
char *strtok (char *s1, const char *s2);
```

Class

Standard C Function

Description

The **strtok** function considers the string *s1* to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string *s2*.

The first call to the **strtok** function returns a pointer to the initial character of the first token, and writes a NULL character into *s1* immediately following the returned token. The function keeps track of its position in the string between separate calls so that subsequent calls (with the first argument a NULL pointer) will work through *s1* immediately following that token until no tokens remain. The separator string *s2* can be different from call to call.

When no tokens remain in *s1*, a NULL pointer is returned.

Notes

The function **strtok** modifies the static variable **_lastp**; no other static data is modified.

strtol — Converts ASCII String to a Long Integer

Syntax

```
#include <stdlib.h>
long strtol (const char *ptr, char **endptr, int base);
```

Class

Standard C Function

Description

The **strtol** function converts an ASCII string pointed to by *ptr* to a `long` integer. This function takes ASCII decimal digits from the left. It ignores leading blanks and tabs. An optional sign (+ or -) can be included to obtain a signed result. After the optional sign, leading zeros are ignored. If *base* is 16, the leading 0x is also ignored. The first unrecognized character ends the conversion.

If *endptr* is not a NULL pointer, a pointer to that character is stored in the object *endptr* points to.

The *base* must be between 2 and 36; otherwise, base 10 is assumed. If *base* is 0, the string itself determines the base in the same manner as an ordinary integer constant.

The **strtol** function returns the converted value.

If the correct value is too large to be represented, LONG_MAX or LONG_MIN, depending on the sign of the value, is returned and the value of the macro ERANGE is stored in **errno**. If the string starts with an unrecognized character, **strtol** returns zero.

Notes

The function **strtol** can modify the static variable **errno**; no other static data is modified.

strtoul — Converts String to Unsigned Long Integer

Syntax

```
#include <stdlib.h>
unsigned long int strtoul (const char *nptr, char **endptr, int base);
```

Class

Standard C Function

Description

The strtoul function converts an ASCII string pointed to by *nptr* to an `unsigned long` integer. This function takes ASCII decimal digits from the left. It ignores leading blanks and tabs. An optional sign (+ or -) can be included to obtain a signed result. After the optional sign, leading zeros are ignored. If *base* is 16, the leading 0x is also ignored. The first unrecognized character ends the conversion.

If *endptr* is not a NULL pointer, a pointer to that character is stored in the object that *endptr* points to.

The *base* must be between 2 and 36; otherwise, base 10 is assumed. If *base* is 0, the string itself determines the base in the same manner as an ordinary integer constant.

The strtoul function returns the converted value.

If the correct value is too large to be represented, ULONG_MAX is returned and the value of the macro ERANGE is stored in errno. If the string starts with an unrecognized character, strtoul returns zero.

Notes

The function strtoul can modify the static variable errno; no other static data is modified.

swab — Swaps Odd and Even Bytes in Memory

Syntax

```
#include <mriext.h>
void swab (char *source, char *dest, int count);
```

Class

Microtec Research C Extension

Description

The swab function moves data from *source* to *dest*, swapping odd and even bytes in the process. The parameter *count* should be even. If *count* is odd, the last byte of *source* is not moved.

Notes

If *source* and *dest* memory spaces overlap, the behavior is undefined.

tan — Computes Tangent of a Number

Syntax

```
#include <math.h>
double tan (double d);
```

Class

Standard C Function

Description

The tan function returns the tangent of *d*. Arguments must be in radians.

Notes

The result is undefined if the argument is out of range.

For information on the single-precision version of this function, tanf, refer to the *mathf.h* section at the beginning of this chapter.

tanh — Computes Hyperbolic Tangent of a Number

Syntax

```
#include <math.h>
double tanh (double d);
```

Class

Standard C Function

Description

The **tanh** function returns the hyperbolic tangent of *d*.

Notes

The result is undefined if the argument is out of range.

For information on the single-precision version of this function, **tanhf**, refer to the *mathf.h* section at the beginning of this chapter.

time — Determines Current Calendar Time

Syntax

```
#include <time.h>
time_t time (time_t *timer);
```

Class

Standard C Function

Description

The time function returns the current calendar time. This time is expressed as the number of seconds since 00:00:00, January 1, 1970.

This particular implementation of time only returns (`time_t`)-1 to indicate that the calendar time is unavailable. No usable timing values are returned.

Notes

This routine is provided as a service to the user only. If you wish to provide your own time function, you must link it in ahead of the libraries.

toascii — Converts a Byte to ASCII Format

Syntax

```
#include <mriext.h>
int toascii (int c);
```

Class

Microtec Research C Extension (implemented as a function and as a macro)

Description

The toascii function converts the value of its argument *c* to an ASCII character by truncating all but the lower-order 7 bits. The argument *c* is not modified. toascii returns a value in the range 0 to 127.

The macro is invoked by default. If you want to call the function, you must either:

- enclose the name in parentheses:

```
(toascii)(c);
```

or

- undefine the `toascii` macro with `#UNDEF` before using it:

```
#UNDEF toascii
```

Notes

None.

tolower — Converts Characters to Lowercase

Syntax

```
#include <ctype.h>
int tolower (int c);
```

Class

Standard C Function

Description

The tolower function converts the value of its argument *c* to a lower-case letter. If *c* is not an upper-case letter, the conversion is not performed. The argument *c* is not modified.

The macro is invoked by default. If you want to call the function, you must either:

- enclose the name in parentheses:

```
(tolower)(c);
```

or

- undefine the `tolower` macro with `#UNDEF` before using it:

```
#UNDEF tolower
```

Notes

If *c* is outside the range -1 to 255, the result is undefined.

_tolower — Converts Characters to Lowercase Without Argument Checking

Syntax

```
#include <mriext.h>
int _tolower (int c);
```

Class

Microtec Research C Extension (implemented as a function and as a macro)

Description

The _tolower function converts the value of its argument *c* to a lower-case letter. The conversion is performed whether or not *c* is an upper-case letter. The argument *c* is not modified.

The macro is invoked by default. If you want to call the function, you must either:

- enclose the name in parentheses:

```
(_tolower)(c);
```

or

- undefine the `_tolower` macro with `#UNDEF` before using it:

```
#UNDEF _tolower
```

Notes

If *c* is not an upper-case letter, the result is undefined.

toupper — Converts Characters to Uppercase

Syntax

```
#include <ctype.h>
int toupper (int c);
```

Class

Standard C Function

Description

The **toupper** function converts the value of its argument *c* to an upper-case letter. If *c* is not a lower-case letter, the conversion is not performed. The argument *c* is not modified.

The macro is invoked by default. If you want to call the function, you must either:

- enclose the name in parentheses:

```
(_toupper)(c);
```

or

- undefine the `_toupper` macro with `#UNDEF` before using it:

```
#UNDEF _toupper
```

Notes

If *c* is outside the range -1 to 255, the result is undefined.

_toupper — Converts Characters to Uppercase Without Argument Checking

Syntax

```
#include <mriext.h>
int _toupper (int c);
```

Class

Microtec Research C Extension (implemented as a function and as a macro)

Description

The _toupper function converts the value of its argument *c* to an upper-case letter. The conversion is performed whether or not *c* is a lower-case letter. The argument *c* is not modified.

The macro is invoked by default. If you want to call the function, you must either:

- enclose the name in parentheses:

```
(_toupper)(c);
```

or

- undefine the `_toupper` macro with `#UNDEF` before using it:

```
#UNDEF _toupper
```

Notes

If *c* is not a lower-case letter, the result is undefined.

ungetc — Returns a Character to a File

Syntax

```
#include <stdio.h>
int ungetc (int c, FILE *file);
```

Class

Standard C Function

Description

The ungetc function returns the current character to an input file after it has been read.

The character *c* is pushed back to the file indicated by *file*. Only one character can be pushed onto the file before it is read from again.

If ungetc is successful, the character *c* is returned.

If ungetc cannot write to the file for any reason, or if any of the conditions listed in the description are not met, ungetc fails and returns EOF (End-Of-File).

Notes

The function ungetc can modify the static array _iob; no other static data is modified.

unlink — Unlinks a File Name

Syntax

```
int unlink (char *filename);
```

Class

System Function

Description

The **unlink** function removes the ability to access the file named by *filename*. Generally, this is accomplished by removing the directory entry for *filename*. If no other links or directory entries for the file exist, then the file is deleted.

This function is implemented as a stub and always returns 0. The **unlink** function should return -1 if the operation fails.

Notes

Since embedded systems do not usually have a file system, the **unlink** function is provided as a stub only; it does nothing and always returns a 0. If it is used, it will generate a link-time error indicating that the **_WARNING_unlink_stub_used** symbol is unresolved. If it is executed, it will write "WARNING - unlink (stub routine) called" to stderr. You must provide your own **unlink** function to allow it to work in your environment.

If you are developing code for an environment that does have a file system, you can provide a replacement for the **unlink** function stub by linking in your implementation of **unlink** before the run-time library.

va_arg — Returns Arguments from a Variable Argument List

Syntax

```
#include <stdarg.h>
type va_arg (va_list ap, type);
```

Class

Standard C Macro (implemented as a macro only)

Description

The va_arg macro obtains the next argument from a function that has a variable number of arguments. It is used in conjunction with va_start and va_end to access any arguments that may have been passed in addition to the ones declared explicitly in the function header.

To access these additional arguments, do the following:

1. Declare a local variable *ap* of type `va_list`. This is an argument pointer that will be used by va_arg to access the next available argument.
2. Call va_start(*ap*,*rightmost_param*), where *rightmost_param* is the name of the last parameter explicitly declared in the function header. This initializes *ap* to refer to the next parameter, if any, following the *rightmost_param*.
3. Call `va_arg(ap,type)` as many times as necessary to sequentially obtain any arguments that may have been passed after *rightmost_param*. The *type* is the expected type of the argument. The *type* must be compatible with the actual next argument as promoted according to the default argument promotions. In other words, sizeof (*type*) >= sizeof (int).
4. Finally, call va_end(*ap*); after obtaining all the arguments.

This example illustrates the use of va_start, va_arg, and va_end to access the arguments of a function. This function prints each of its "..." arguments; the first argument indicates how many "..." arguments have been passed.

```
#include <stdarg.h>
#include <stdio.h>
void print_args (unsigned arg_count, ...)
{
   va_list ap;
   va_start (ap, arg_count);
   while (arg_count--) puts (va_arg (ap, char *));
   va_end (ap);
}
```

Notes

If va_arg is used after all the available arguments have been obtained, the result is undefined. If *type* is not compatible with the actual next argument as promoted according to the default argument promotions, the behavior is undefined.

va_end — Terminates Access to a Variable Argument List

Syntax

```
#include <stdarg.h>
void va_end (va_list ap);
```

Class

Standard C Macro

Description

The va_end macro terminates access to a variable argument list. The macro must be invoked before the return from the function containing the corresponding va_start.

Notes

If there is no corresponding invocation of the va_start macro, or if the va_end macro is not invoked before the return, the behavior is undefined.

va_start — Allows Access to a Variable Argument List

Syntax

```
#include <stdarg.h>
void va_start (va_list ap, parmN);
```

Class

Standard C Macro (implemented as a macro only)

Description

The macro va_start initiates access to a variable argument list. It must be invoked before any access to unnamed arguments. It initializes *ap* for access by va_arg and va_end. *parmN* is the rightmost parameter in the variable parameter list in the function definition (the one just before the ,...).

va_start must be invoked in the function accepting a varying number of arguments.

Notes

If *parmN* is declared with the register storage class, or with a function or array type, the behavior is undefined. If *parmN* is declared with a type that is not compatible with the type that results after application of the default argument promotions, the behavior is undefined. To be safe, define *parmN* such that sizeof(*parmN*) >= sizeof(int).

vfprintf — Provides Formatted Print to a Specified File

Syntax

```
#include <stdarg.h>
#include <stdio.h>
int vfprintf (FILE *stream, const char *format, va_list arg);
```

Class

Standard C Function

Description

The **vfprintf** function is equivalent to **fprintf** with the variable argument list replaced by *arg*. The value of *arg* must have been set by the **va_start** macro and can be modified by subsequent **va_arg** calls. The **vfprintf** function does not invoke the **va_end** macro.

The **vfprintf** function returns the number of characters transmitted to a file if successful or a negative value if an output error occurred.

Notes

The **vprintf** function can modify the static array **_iob**; no other static data is modified.

vprintf — Provides Formatted Print on Standard Output

Syntax

```
#include <stdarg.h>
#include <stdio.h>
int vprintf (const char *format, va_list arg);
```

Class

Standard C Function

Description

The **vprintf** function is equivalent to **printf**, with the variable argument list replaced by *arg*. The value of *arg* must have been set by the **va_start** macro and can be modified by subsequent va_arg calls. The **vprintf** function does not invoke the **va_end** macro.

The **vprintf** function returns the number of characters transmitted or a negative value if an output error occurred.

Notes

The **vprintf** function can modify the static array _iob; no other static data is modified.

vsprintf — Formats a String

Syntax

```
#include <stdarg.h>
#include <stdio.h>
int vsprintf(char *s, const char *format, va_list arg);
```

Class

Standard C Function

Description

The vsprintf function is equivalent to sprintf, with the variable argument list replaced by *arg*. The value of *arg* must have been set by the va_start macro and possibly modified by subsequent va_arg calls. The vsprintf function does not invoke the va_end macro.

The vsprintf function returns the number of characters transmitted or a negative value if an output error occurred.

Notes

None.

write — Writes Bytes to a Specified File

Syntax

```
int write (int fd, char *buffer, int nbyte);
```

Class

System Function

Description

The write function writes *nbyte* bytes from a buffer to the file associated with the file descriptor *fd*. The starting buffer address is specified by *buffer*. The file descriptor *fd* is returned by open or creat. Up to 64K bytes may be written, starting at the current file position.

The write function returns the number of bytes actually written, or a -1 if an error occurs.

Notes

Since embedded systems do not usually have a file system, the write function provided in the run-time library ignores the file descriptor *fd*, and writes to the address of the variable _simulated_output. The source for the write function is provided on the distribution. If you want to modify the write function or replace it with one of your own, you should link in your implementation of write before the run-time library.

zalloc — Allocates Data Space Dynamically

Syntax

```
include <mriext.h>
void *zalloc (size_t *size)
```

Class

Microtec Research C Extension

Description

The zalloc function allocates a new area of memory for program use and returns a pointer to that area. The size of the area allocated is equal to *size* bytes. The zalloc function sets the area to zeros. The space allocated is guaranteed to start on a boundary that is suitable for aligning any type.

If the requested amount of space cannot be allocated, zalloc returns a NULL pointer. Allocations are provided from a data structure called the heap, which is located in the stack section.

Notes

The function zalloc can modify the static struct _membase, or the static variables _avail or _badlist; no other static data is modified.

Internal Data Representation 6

Overview

This chapter describes internal data formats of the C language and the run-time organization of C programs for the 68000 family microprocessor-based systems.

Storage Layout

The illustrations below show the conventions for storage layout. Figures 6-1 and 6-2 illustrate how words and long words are ordered from lower to higher address, top to bottom.

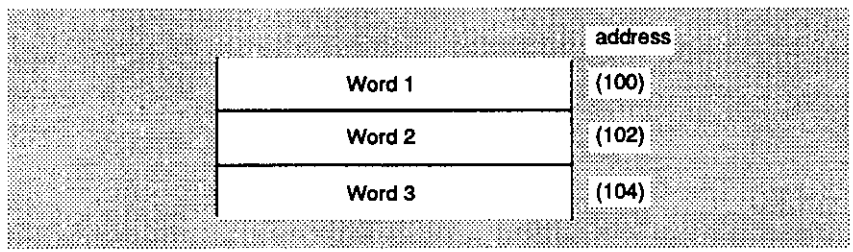

Figure 6-1. Word Ordering

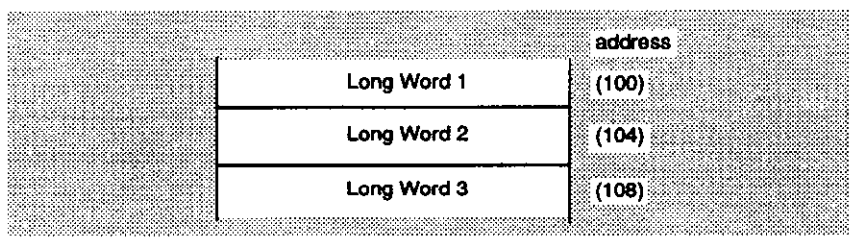

Figure 6-2. Long Word Ordering

Figures 6-3 and 6-4 illustrate how the bytes in a word and long word are ordered from lower to higher address, left to right.

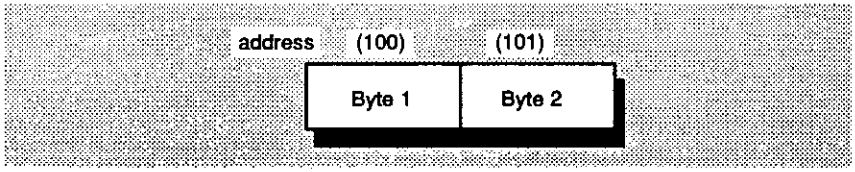

Figure 6-3. Byte Ordering in Words

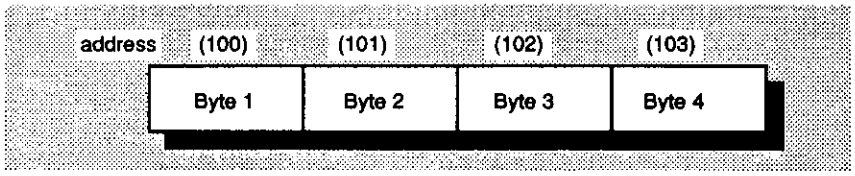

Figure 6-4. Byte Ordering in Long Words

Figure 6-5 illustrates the position in the word of the most significant bit and the least significant bit. For words, the least significant bit is bit 0 and the most significant bit is bit 15. For long words, the least significant bit is bit 0 and the most significant bit is bit 31.

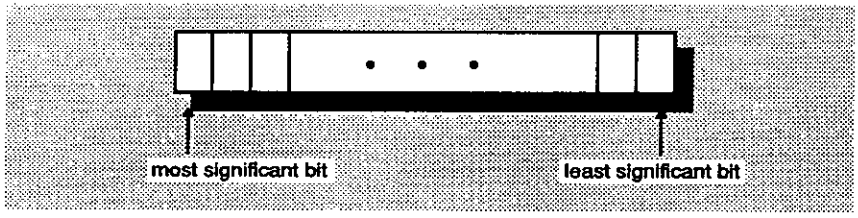

Figure 6-5. Position of the Most and Least Significant Bits

Data Type Summary

The size and value range of the basic data types are shown in Tables 6-1 and 6-2. The range of values are decimal representations, and the alignment is in bytes.

Table 6-1. Data Types

Data Type	Size	Range	68000/010 /CPU32 Alignment	68020 /030/040 Alignment
char	8 bits = 1 byte	-128 to 127	1	1
unsigned char	8 bits = 1 byte	0 to 255	1	1
short int	16 bits = 2 bytes	-32768 to 32767	2	2
unsigned short int	16 bits = 2 bytes	0 to 65535	2	2
enum	32 bits = 4 bytes		2	4
int	32 bits = 4 bytes	-2147483648 to 2147483647	2	4
unsigned int	32 bits = 4 bytes	0 to 4294967295	2	4
pointer	32 bits = 4 bytes		2	4
long int	32 bits = 4 bytes	-2147483648 to 2147483647	2	4
unsigned long int	32 bits = 4 bytes	0 to 4294967295	2	4
float	32 bits = 4 bytes	(see *Float* section)	2	4
double	64 bits = 8 bytes	(see *Double and Long Double* section)	2	4
long double	64 bits = 8 bytes	(see *Double and Long Double* section)	2	4

Table 6-2. Complex Types

Complex Types	Size
array	Combined size of elements
struct	Combined size of members (plus possible padding)
union	Size of largest member

Scalar Data Types

The scalar data types in the C language are **char**, **int**, **float**, and **double**. In Microtec Research C, **int** types have two sizes: **short** (16-bit) and **long** (32-bit). The default size is **long**. An **int** type can be signed or unsigned. It is signed if the keyword **unsigned** does not appear. If you declare an **int** with the keywords **short**, **long**, or **unsigned**, you can omit the keyword **int**. The compiler represents all **int** data in two's complement form. With multi-byte storage, the most significant byte is stored at the lowest memory address. Each scalar data type is described below.

char

The **char** type is a signed quantity used to store single ASCII characters. A variable of type **char** is one byte (8-bits) in size and has a range of -128 to 127. The internal representation is shown in Table 6-3. Bit 7 is the sign field; bits 6 through 0 contain the value. Some sample **char** values are shown in this table with the sign (S) and value fields; the sign field is shown in bold.

Table 6-3. Representation of char

char	Representation in Bits
	S Value
	Bits: 7654 3210
0	**0**000 0000
1	**0**000 0001
-1	**1**111 1111
'a'	**0**110 0001

If the keyword **unsigned** does not appear, the **char** will be considered to be **signed**, unless the /**unsignedchar** (VMS) or -**Ku** (UNIX/DOS) option is used. In this case, it will be considered to be **unsigned**.

unsigned char

The **unsigned char** type is an unsigned quantity used to store a single ASCII character. A variable of type **unsigned char** is one byte in size and has a range of 0 to 255. The internal representation is shown in Table 6-4. Bits 7 through 0 contain the value. Some sample **unsigned char** values are shown in this table.

Table 6-4. Representation of unsigned char

unsigned char	Representation in Bits	
	Value	
Bits:	7654	3210
0	0000	0000
1	0000	0001
255	1111	1111
'a'	0110	0001

short int

The **short int** type is a signed quantity used to store positive and negative integer values. You can declare it with the keyword **short**. A variable of type **short int** is two bytes in size and has a range of -32768 to 32767. The internal representation is shown in Table 6-5. Bit 15 is the sign field; bits 14 through 0 contain the value. Some sample **short int** values are shown in this table with the sign (S) and value fields; the sign field is shown in bold.

Table 6-5. Representation of short int

short int	Representation in Bits			
	S	Value		
	1	1		
Bits:	5432	1098	7654	3210
0	0000	0000	0000	0000
1	0000	0000	0000	0001
-1	1111	1111	1111	1111
23500	0101	1011	1100	1100

unsigned short int

The **unsigned short int** type is an unsigned quantity used to store only non-negative integer values. You can declare it with the keywords **unsigned short**. A variable of type **unsigned short int** is two bytes in size and has a range of 0 to 65535. The internal representation is shown in Table 6-6. Bits 15 through 0 contain the value. Some sample **unsigned short int** values are shown in this table.

Table 6-6. Representation of unsigned short int

unsigned short int	Representation in Bits			
	Value			
	1 5	1 0 9 8	7 6 5 4	3 2 1 0
Bits:				
0	0000	0000	0000	0000
1	0000	0000	0000	0001
65535	1111	1111	1111	1111
23500	0101	1011	1100	1100

int

The **int** type is a signed quantity used to store positive and negative integer values. You can declare it with the keyword **int**. A variable of type **int** is four bytes in size and has a range of -2147483648 to 2147483647.

> **Note**
>
> The range of values for an **int** variable is the same as for a **long int**.

The internal representation is shown in Table 6-7. Bit 31 contains the sign; bits 30 through 0 contain the value. Some sample **int** values are shown in this table with the sign (S) and value fields; the sign field is shown in bold.

Table 6-7. Representation of int

int	Representation in Bits							
	S	Value						
	3	2	2	1	1	1		
Bits:	1098	7654	3210	9876	5432	1098	7654	3210
0	0000	0000	0000	0000	0000	0000	0000	0000
1	0000	0000	0000	0000	0000	0000	0000	0001
-1	1111	1111	1111	1111	1111	1111	1111	1111
44120345	0000	0010	1010	0001	0011	1001	0001	1001

unsigned int

The unsigned int type is an unsigned quantity used to store non-negative integer values. You can declare it with the keyword unsigned. A variable of type unsigned int is four bytes in size and has a range of 0 to 4294967295.

Note

The range of values for an unsigned int variable is the same as for an unsigned long int variable.

The internal representation is shown in Table 6-8. Bits 31 through 0 contain the value. Some sample unsigned int values are shown in this table.

Table 6-8. Representation of unsigned int

unsigned int	Representation in Bits							
	Value							
	3	2	2	1	1	1		
Bits:	1098	7654	3210	9876	5432	1098	7654	3210
0	0000	0000	0000	0000	0000	0000	0000	0000
1	0000	0000	0000	0000	0000	0000	0000	0001
4294967295	1111	1111	1111	1111	1111	1111	1111	1111
44120345	0000	0010	1010	0001	0011	1001	0001	1001

long int

The long int type is a signed quantity used to store positive and negative integer values. You can declare it with the keyword long. A variable of type long int is four bytes in size and has a range of -2147483648 to 2147483647. The internal representation is shown in Table 6-9. Bit 31 contains the sign; bits 30 through 0 contain the value. Some sample long int values are shown in this table with the sign (S) and value fields; the sign field is shown in bold.

Table 6-9. Representation of long int

int	Representation in Bits							
	S	Value						
	3	2	2	1	1	1		
Bits:	1098	7654	3210	9876	5432	1098	7654	3210
0	**0**000	0000	0000	0000	0000	0000	0000	0000
1	**0**000	0000	0000	0000	0000	0000	0000	0001
-1	**1**111	1111	1111	1111	1111	1111	1111	1111
44120345	**0**000	0010	1010	0001	0011	1001	0001	1001

unsigned long int

The unsigned long int type is an unsigned quantity used to store only non-negative integer values. You can declare it with the keywords unsigned long. A variable of type unsigned long int is four bytes in size and has a range of 0 to 4294967295. The internal representation is shown in Table 6-10. Bits 31 through 0 contain the value. Some sample unsigned long int values are shown in this table.

Table 6-10. Representation of unsigned long int

unsigned long int	Representation in Bits							
	Value							
	3	2	2	1	1	1		
Bits:	1098	7654	3210	9876	5432	1098	7654	3210
0	0000	0000	0000	0000	0000	0000	0000	0000
1	0000	0000	0000	0000	0000	0000	0000	0001
4294967295	1111	1111	1111	1111	1111	1111	1111	1111
44120345	0000	0010	1010	0001	0011	1001	0001	1001

Data Types — Float and Double

Microtec Research C stores floating-point data types using the IEEE single and double-precision format. The formats contain a sign bit field, an exponent field, and a fraction field.

- Sign Bit Field

  The sign of the number being represented is stored in the sign bit field. If the number is positive, the sign bit field contains the value 0. If it is negative, it contains the value 1. The sign bit field is stored in the most significant bit of a floating-point value.

- Exponent Field

  The exponent of a number is multiplied by its fractional part to get a value. The exponent can store a greater range of values than could normally be stored. The exponent field of the number contains a "biased" form of the exponent. The bias is subtracted from the exponent field to get the actual exponent. This allows both positive and negative exponents. The value of the bias is different for float and double types.

  Note

  If both the exponent field and the fraction field are equal to zero, the number being represented will also be zero.

- Fraction Field

  IEEE floating-point format stores the fractional part of a number in a "normalized" form. It assumes that all non-zero numbers are of the following form:

  1.*xxxxxx* (binary)

  The character *x* represents either 0 or 1 (binary).

  Since all floating-point numbers begin with 1, the 1 becomes the implicit normalized bit. It is the most significant bit of the fraction and is not stored in memory. The binary point is located immediately to the right of the normalized bit. All bits after the binary point represent values less than 1 (binary).

Example:

The number 1.625 (decimal) can be represented as:

1.101 (binary)

This number can be represented as:

$2^0 + 2^{-1} + 2^{-3}$ (decimal)

or

1 + 0.5 + 0.125 (decimal)

or

1.625 (decimal)

- The Represented Value

  The value of the number being represented is equal to the exponent multiplied by the fractional part, with the sign specified by the sign bit field.

Float

The single-precision form uses four consecutive bytes, with the 32 bits containing a 1-bit sign field (bit 31), an 8-bit biased exponent field (bits 30 through 23), and a 23-bit fraction field (bits 22 through 0). The exponent has a bias of 7F (hexadecimal). The fraction field is precise to 7 decimal digits. The single-precision format can represent values in the range $1.18 * 10^{-38}$ to $3.4 * 10^{38}$ (decimal). The internal representation is shown in Table 6-11. Some sample float values are shown in this table with the sign (S), exponent, and fraction fields; the exponent field is shown in bold.

Table 6-11. Representation of Single-Precision float

float	Representation in Bits							
	S	Exponent		Fraction				
	3	2	2	1	1	1		
Bits:	1098	7654	3210	9876	5432	1098	7654	3210
0.0	**0000**	**0000**	0000	0000	0000	0000	0000	0000
1.0	**0011**	**1111**	1000	0000	0000	0000	0000	0000
NAN (illegal)	**1111**	**1111**	1111	1111	1111	1111	1111	1111
0.75	**0011**	**1111**	0100	0000	0000	0000	0000	0000

Example:

As an example, the value 1.0 in Table 6-11 is calculated as follows:

1. The sign of the value is positive because the sign bit field (bit 31) is equal to 0.
2. The exponent field is equal to 7F (hexadecimal). The actual exponent is calculated by subtracting the bias value (7F) from the exponent field value. The result is 0 (decimal):

   $$7F - 7F = 0$$

   The exponent multiplier is equal to 2^0, which is equal to 1 (decimal).
3. The fraction field is equal to .0 (binary). After adding the implicit normalized bit, the fraction is equal to 1.0 (binary). The fraction value is equal to 2^0 (decimal), which is equal to 1 (decimal).
4. The value of the number is positive $1 \bullet 1 = 1.0$ (i.e., $1.0 * 10^0$).

Example:

As an example, the value 0.75 in Table 6-11 is calculated as follows:

1. The sign of the value is positive because the sign bit field is equal to 0.
2. The exponent field is equal to 7E (hexadecimal). The actual exponent is calculated by subtracting the bias value (7F) from the exponent field value. The result is -1 (decimal):

   $$7E - 7F = -1$$

   The exponent multiplier is equal to 2^{-1}, which is equal to 0.5 (decimal).
3. The fraction field is equal to .1 (binary). After adding the implicit normalized bit, the fraction is equal to 1.1 (binary). The fraction value is equal to $2^0 + 2^{-1}$ (decimal), which is equal to 1 + 0.5 (decimal), which is equal to 1.5 (decimal).
4. The value of the number is positive $0.5 * 1.5 = 0.75$ (i.e., $7.5 * 10^{-1}$).

Double and Long Double

Both the double and long double precision format uses eight consecutive bytes, with the 64 bits containing a sign bit field (bit 63), an 11-bit biased exponent field (bits 62 through 52), and a 52-bit fraction field (bits 51 through 0). The exponent has a bias of 3FF (hexadecimal). The fraction field is precise to 15 decimal digits.

The double-precision format can represent values in the range $2.23 * 10^{-308}$ to $1.79 * 10^{308}$ (decimal). The internal representation is shown in Table 6-12. Some sample double-precision values are shown in this table with the sign (S), exponent, and fraction fields; the exponent field is shown in bold.

Table 6-12. Representation of double or long double

double	Representation in Bits									
	S	Exponent		Fraction						
	6	5	5	5	4	4	...	1		
Bits:	3210	9876	5432	1098	7654	3210	...	1098	7654	3210
0.0	**0000**	**0000**	**0000**	0000	0000	0000	...	0000	0000	0000
1.0	**0011**	**1111**	**1111**	0000	0000	1000	...	0000	0000	0000
NAN (illegal)	**1111**	**1111**	**1111**	1111	1111	0000	...	1111	1111	1111
0.75	**0011**	**1111**	**1110**	1000	0000	1100	...	0000	0000	0000

Example:

As an example, the value 1.0 in Table 6-12 is calculated as follows:

1. The sign of the value is positive because the sign bit field is equal to 0.
2. The exponent field is equal to 3FF (hexadecimal). The actual exponent is calculated by subtracting the bias value (3FF) from the exponent field value. The result is 0 (decimal).

   $$3FF - 3FF = 0$$

   The exponent multiplier is equal to 2^0, which is equal to 1 (decimal).
3. The fraction field is equal to .0 (binary). After adding the implicit normalized bit, the fraction is equal to 1.0 (binary). The fraction value is equal to 2^0 (decimal), which is equal to 1 (decimal).
4. The value of the number is positive $1 * 1 = 1.0$ (i.e., $1.0 * 10^0$).

Example:

As an example, the value 0.75 in Table 6-12 is calculated as follows:

1. The sign of the value is positive because the sign bit field is equal to 0.
2. The exponent field is equal to 3FE (hexadecimal). The actual exponent is calculated by subtracting the bias value (3FF) from the exponent field value. The result is -1 (decimal).

   3FE - 3FF = -1

   The exponent multiplier is equal to 2^{-1}, which is equal to 0.5 (decimal).

3. The fraction field is equal to .1 (binary). After adding the implicit normalized bit, the fraction is equal to 1.1 (binary). The fraction value is equal to $2^{0} + 2^{-1}$ (decimal), which is equal to 1 + 0.5 (decimal), which is equal to 1.5 (decimal).
4. The value of the number is positive $0.5 * 1.5 = 0.75$ (i.e., $7.5 * 10^{-1}$).

Pointers

A pointer is a value corresponding to the address of an object in memory. A pointer can point to an object of any data type. Pointers occupy four bytes and are aligned on four-byte boundaries.

A pointer variable is declared by placing the pointer operator (*) in front of a variable name. The following are examples of variables declared as pointers.

```
char *pc;          /* pc is a pointer to a char  */
int  *ptr;         /* ptr is a pointer to an int */
```

A pointer variable stores the address of an object it points to in memory. To obtain the address of an object, use the "address of" operator (&).

You can declare pointers to any type, including functions. In particular, if more than one pointer operator (*) precedes a variable name, the variable is declared as a pointer to a pointer. For example:

```
char **ppc;        /* ppc is a pointer to a pointer to a */
                   /* char */
int  **ptr_ptr;    /* ptr_ptr is a pointer to a pointer to */
                   /* an int */
```

In Microtec Research C, the type pointed to must have been previously declared.

Bit Fields

A member of a structure can be defined as a bit field. A bit field defines the number of bits of storage that the member requires. A bit field is specified by:

basetype member_name:*number_of_bits*;

In addition to base types signed int and unsigned int, Microtec Research C also supports char, short, and long (signed and unsigned). If the keyword unsigned is absent, the bit field is signed.The maximum size of a single bit field is the number of bits in its base type.

Example:

```
struct packet {unsigned control:3; int data:13} pvar;
```

Bit fields are stored in bytes starting at higher order bits. If a successive field element does not fit into the remaining space of the current base type, a new byte, word, or long word is allocated (according to the base type). The field member is placed into it starting again from the most significant order bit. Field members of zero length are not allocated any space and cause the next member to be aligned at the next byte, word, or long word (according to the base type of the next member). Bit fields can be initialized.

Changing the sign in a sequence of bit field members does not force any alignment of the next bit field.

Example:

```
struct {
   char bf1 : 5;    /* changing type, pad to next word */
   short bf2 : 3;   /* changing type, pad to next long word */
   int bf3 : 2;
   signed int bf4 : 7;
} s1;
```

Figure 6-6 shows the bit field allocation within the byte for this example.

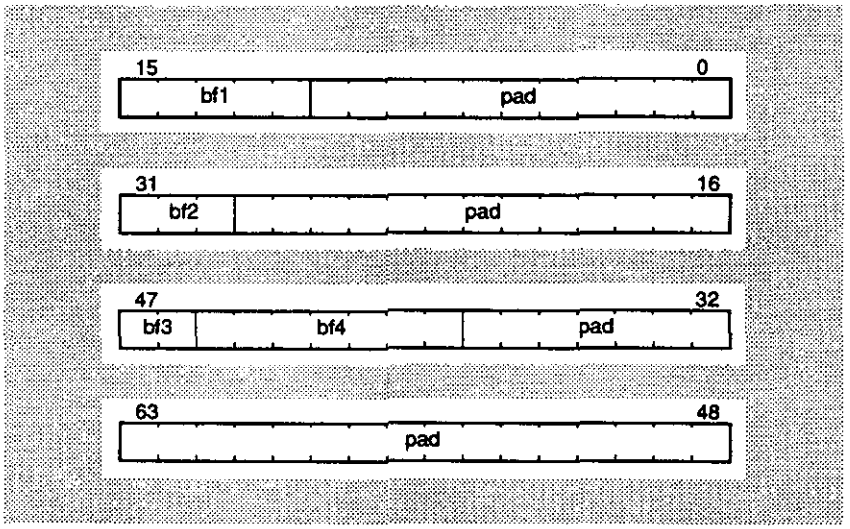

Figure 6-6. Sample Bit Field Allocation (over 4 bytes)

Example:

```
struct {
        char bf1 : 3;
        unsigned char bf2 : 3;      /* changed sign */
        char bf3 : 3;          /* will not fit in current byte */
} s2;
```

Figure 6-7 shows the bit field allocation within the byte for this example.

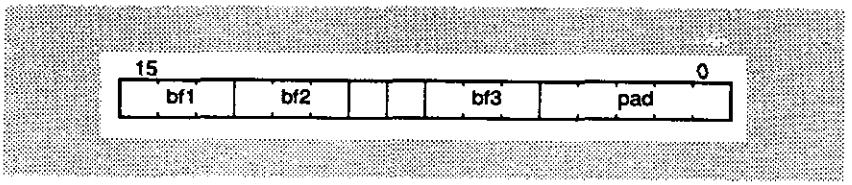

Figure 6-7. Sample Bit Field Allocation (over 2 bytes)

Enumeration Types

Enumeration (enum) types are int types with named constants. Enumeration types take an enumeration list that consists of identifiers declared as constants. The enumeration list can appear wherever constants are required. The keyword enum is used to declare an enumeration type.

The syntax of an enum type is shown below.

enumeration-type-specifier:
enumeration-type-definition
enumeration-type-reference

enumeration-type-definition
enum *enumeration-tag (optional)*
{ *enumeration-definition-list* }

enumeration-type-reference:
enum *enumeration-tag*

For a complete description of the ANSI C syntax, see Appendix B.

The enumerated type name can be the same as the name of a variable in a scope. However, it cannot be the same as the name of any struct or union in the scope. Each enumeration declaration declares a scalar constant of the enumeration type.

If a constant expression appears in an enumeration declaration, it specifies the ordinal value of the identifier. If a constant expression is not given in an enumeration declaration, the ordinal value of the identifier is one greater than the value of the previous enumeration declaration. However, if the enumeration declaration is the first in an enumeration list, its ordinal value is zero. By default, enumeration types are signed.

For more information on the packed enumeration type, refer to the section *Packed Enumeration Types* in this chapter.

Examples:

```
enum weekday {sun, mon, tue, wed, thu, fri, sat} wday;
      /* wday is a variable of type weekday; the enum-specifier
         includes everything except 'wday'. The numeric
         representations are not specified; they will be chosen by
         the compiler */

enum {first=10, second=5, third=3, fourth=2, fifth=1} rankpoints;
      /* Here the numeric representations are specified */
```

Arrays

The array data type is composed of a fixed number of data elements that are all of the same data type. Any data type, including pointers, can be used to declare an array.

Arrays can be single-dimensional and multi-dimensional.

Examples:

```
char  d[25];                /* 25 element array of */
                            /* characters (0..24) */

int   i[15];                /* 15 element array of */
                            /* integers (0..14) */

char  classes[10][10];      /* 100 element array of */
                            /* characters */

int   teachers[100][30];    /* 3000 element array of */
                            /* integers */
```

With two-dimensional arrays, the first subscript ([ ]) corresponds to the row and the second to the column at which an element is stored. The elements are stored in row-major order (the rightmost subscript changes for each successive memory location).

The elements of an array are accessed by specifying a subscript after the array variable name. The first element of an array has a subscript of 0. The last element of an array has a subscript of n-1, where n is the number of elements in the array. The following examples reference the elements of array `i`; this array contains 15 elements.

Examples:

`i[0]`	Accesses the first element in the variable array `i`.
`i[6]`	Accesses the seventh element in the variable array `i`.
`i[14]`	Accesses the last element in the variable array `i`.

When an array name is referenced without a subscript, the result is a pointer to the beginning of the array. This is equivalent to the address of the first element of the array.

The storage requirement of an array is the sum of the space used by its elements. For example, a 20 element array of int uses 20*4=80 bytes. The following program example shows a simple use of arrays:

```
main()
{
   int ary[20];   /* automatically allocated when 'main' */
                  /* is entered.*/
   ary[1] = 1;
}
```

Strings

Strings are arrays of character data. Each character occupies one array position. A string enclosed in double quotation marks (" ") contains the number of characters in the string, plus one more for the NULL (\0) character.

Example:

```
char str[] = "this is a string";
```

The above string, `str`, occupies 17 character positions. The actual printed characters occupy 13 positions, the blank spaces between words occupy three positions, and the terminating NULL character occupies one position.

Structures

Structures define a single variable that is actually composed of several variables. The actual variables in a structure are called members, and the members are referenced by using the single variable name as the prefix. Members of a structure do not have to be of the same data type.

structure-type-specifier:
structure-type-definition
structure-type-reference

For a complete description of the ANSI C syntax, see Appendix B.

Unlike the pre-ANSI definition of C, Microtec Research C allows structures to be assigned, passed as parameters to functions, and returned from functions. Structure values are placed on the stack.

The amount of storage needed by a structure is the sum of the space used by its members plus padding.

The alignment of structure members depends on the processor. In 68000/010/CPU32 mode, non-char members will be even-aligned. There may be one byte of padding at the end of the structure to make it even-sized. In 68020/030/040 mode, 4-byte members will be quad-aligned and 2-byte members will be even-aligned. There may be padding at the end of the structure to make it quad-sized.

For example, in 68000/010/CPU32 mode, a structure of the following type is allocated 4 + 1 + (1 byte padding) + 4 = 10 bytes. In 68020/030/040 mode, the structure is allocated 4 + 1 + (3 byte padding) + 4 = 12 bytes.

Example:

```
struct date   /* structure tag */

{
   int day;              /* structure member type int */
   unsigned char month; /* structure member type */
                         /* unsigned char */
   int year;             /* structure member type int */
} holiday;               /* structure variable name */

main()
{  holiday.day = 25;
   holiday.month = 12;
   holiday.year = 1988;
}
```

These rules can be modified with the /align=2 and /align=4 (VMS) or -Z2 and -Z4 (UNIX/DOS) options.

In the example above, the structure variable is `holiday` and the members of the structure are `day`, `month`, and `year`. The `date` is an identifier called a structure tag. This structure tag can be used for later definitions and declarations without repeating the structure members. For example:

```
struct date workday;
```

For more information on packed structures, refer to the section titled *Packed Structures*.

Unions

A union is a variable that holds objects of different types and sizes, but only one at a time. Its declaration resembles that of a structure variable, but the meaning is different: all members of a union share a common area of memory. The union variable is allocated the same space as its largest element plus padding. In 68000/010/CPU32 mode, there may be one byte of padding at the end of the union to make it even-sized. The syntax for a *union-type-specifier* is as follows:

union-type-specifier:
union-type-definition
union-type-reference

For a complete description of the ANSI C syntax, see Appendix B.

Example:

```
union u_tag                 /* union variable tag       */
{
      int ival;             /* union member type int    */
      float fval;           /* union member type float  */
      char *pvar;           /* union member type char*  */
      double dval;          /* union member type double */
}
uval;                       /* union variable name      */


main()
{
      uval.ival = 10;       /* Store 10 in the ival   */
                            /* member of uval.        */
                            /* This affects all other */
                            /* union members since    */
                            /* all members share the  */
                            /* same memory area.      */
}
```

The union provides a way to access the same area of memory in different ways.

Alignment and Packing

This section describes the alignment and packing of bit fields.

Packing Bit Fields

Bit fields are packed into a long word from left (most significant bit) to right (least significant bit). Each field is packed into a single long word. If there is not enough room for the next bit field, it is allocated to a new long word.

Alignment of Bit Fields

A long word which contains bit fields is aligned to a 2-byte boundary in 68000/010/CPU32 mode and a 4-byte boundary in 68020/030/040 mode.

If the next bit field does not fit in the current long word, it will be allocated in the next long word. This is illustrated in Figure 6-8.

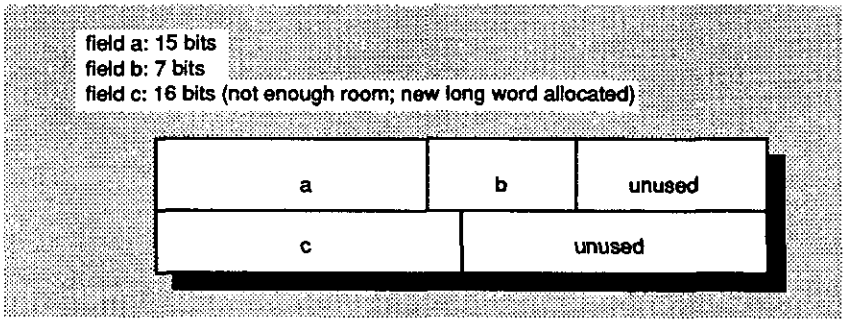

Figure 6-8. Packing Bit Fields

Specifying a bit field with length 0 forces the next bit field to be allocated in the next word.

Both signed and unsigned bit fields are implemented. Signed bit fields should be handled like signed data. Changing the sign in a sequence of bit field members does not introduce a new word. This is illustrated in Figure 6-9.

For more information on bit field base types, refer to the section *Bit Fields*.

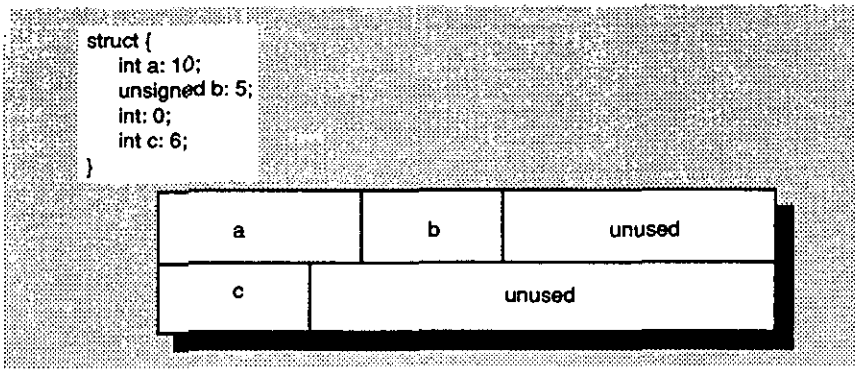

Figure 6-9. Signed Bit Fields

The compiler can support bit fields of **short** (word) and **char** (byte) sizes.

Aggregates

In aggregates such as array, structures, and unions, each member is aligned to the boundary of its original alignment. The aggregate itself takes the alignment of the member with the maximum alignment size.

Examples (68000/010/CPU32):

array of **double**	(2-byte alignment)
struct with long word and byte member	(2-byte alignment)
union with word and byte member	(2-byte alignment)
array of **char**	(1-byte alignment)

Examples (68020/030/040):

array of **double**	(4-byte alignment)
struct with long word and byte member	(4-byte alignment)
union with word and byte member	(2-byte alignment)
array of **char**	(1-byte alignment)

Size of Aggregates

The size of an aggregate should be divisible by its alignment size. There can be unused space after the last member. This unused space should not be occupied by any effective data. This is shown in Figure 6-10.

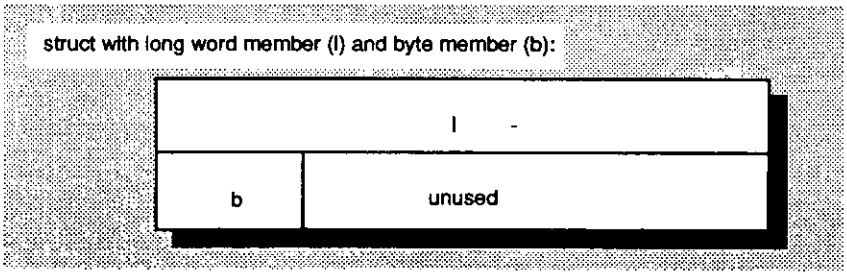

Figure 6-10. Packing of Aggregates

The sizeof operator applied to an aggregate always returns a number divisible by its alignment size.

Examples:

```
sizeof (struct{int a; char c;})          ⟶  8  (4-byte alignment)
sizeof(struct{short a, b; char c;})      ⟶  6  (2-byte alignment)
sizeof(char [3][7])                      ⟶  21 (1-byte alignment)
```

Packed Structures

By default, structure members will be aligned as described in the section titled *Structures*. The packed keyword can be used to force no padding between structure members. In 68000/010/CPU32 mode, there may be 1 byte of padding at the end of a packed structure in order to make the structure even-sized. In 68020/030/040 mode, there will be no padding.

The following example shows the allocation for a packed and unpacked structure member in 68000/010/CPU32 mode and 68020/030/040 mode.

Example:

```
/* PKD will be defined to be packed or unpacked */
#define PKD packed
typedef PKD struct {
        char c1;
        int i;
        char c2;
        char c3;
} str;
```

Figure 6-11 illustrates the unpacked and packed structures.

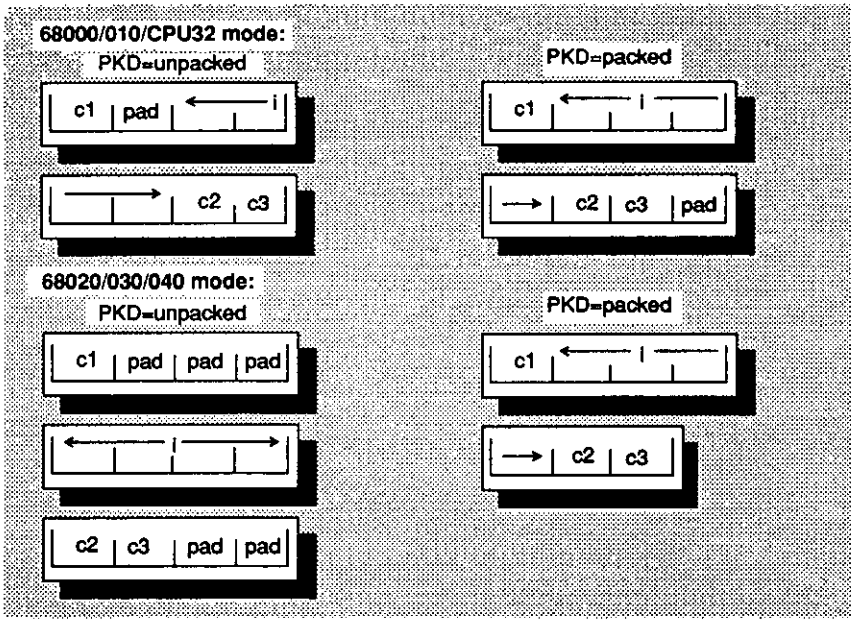

Figure 6-11. Unpacked and Packed Structures

Packed Enumerator Types

A standard unpacked enumerated data type can be said to have signed int as an underlying type. A packed enumerated instead is given an underlying integral type that is chosen to minimize the amount of storage necessary to efficiently hold its values. The underlying integral type will be unsigned if there is at least an unsigned enumerated constant (e.g., 0u,0xffffffff) in the enumeration definition; otherwise, it will be signed.

Examples:

	Underlying Type
`enum color {red, yellow, green};`	`signed int`
`unpacked enum color {red, yellow, green};`	`signed int`
`packed enum color {red, yellow, green};`	`signed char`
`packed enum color {red=1, yellow, green=100};`	`signed char`
`packed enum color {red=127, yellow, green};`	`signed short`
`packed enum color {red=1, yellow, green=127+1};`	`signed short`
`packed enum color {red=0u, yellow, purple=255};`	`unsigned char`
`packed enum color {green=MAX_UNSIGNED_INT};`	`unsigned int`
`packed enum color {x=32768};`	`signed int`
`packed enum color {x=100000U};`	`unsigned int`

Allocation of Variables

Memory variables are allocated based on their size and frequency of use. Scalar and pointer variables generally qualify for allocation to a register unless the variable address is taken. Floating-point variables qualify for allocation to a register if a floating-point coprocessor is available.

Local and static variables are discussed below.

Local Variables

Local (automatic) variables can be put into registers by the compiler even if a register declaration is not used. Eight, sixteen, and thirty-two bit variables can be allocated into registers; however, only the low order bits are valid; the high order bits are undefined.

The lifetime of each local variable is studied and, if possible, several variables may share the same register in one routine. When a variable is allocated to a register, it will always reside in that register. However, since other variables may share the register, the value of the register may not always contain the value of the variable.

Variables that are declared in register declarations are allocated to registers before non-register variables. This allows variables to be specified in their order of importance.

Static Variables

According to the C language definition, all static variables are initialized to zero unless they are explicitly initialized.

Static variables can be allocated to registers if they are assigned in subroutines before they are referenced. These variables can be reorganized by the compiler, but memory may not be allocated to variables that are never used.

By default, uninitialized static variables will be allocated in the zerovars section. This section is cleared at program start-up time. For options that can alter this behavior, see *Initialize Uninitialized Global Data* in the *User's Guide*.

Memory Allocation

Memory is byte addressed with the least significant byte at the lowest address (i.e., high to low ordered). Bits are numbered with bit zero as the least significant bit.

In 68000/010/CPU32 mode, data items that are larger than 1 byte are even-aligned. The stack and all complex data types are even-aligned. In 68020/030/040 mode, data items that are a multiple of 4 bytes are quad-aligned. Data items that are a multiple of 2 bytes are even-aligned. The stack and all complex data types are quad-aligned.

Packed fields are allocated starting at the most significant bit and must be fully contained in no more than four bytes.

Warning

Programs that rely on byte ordering (that is, programs that declare a variable **int** in one module and **char** in another) might not work.

C Calling Conventions 7

Overview

This chapter describes C and assembly language calling conventions for the Microtec Research C compiler. This chapter provides enough information for you to write assembly language routines to interface with C language functions.

Parameter Passing

Assembly language routines must follow the C parameter passing conventions to pass values to, or to receive values from, C functions. C functions pass parameters by value on the stack. Parameters are always evaluated from right to left, with the first parameter being evaluated last. The next section describes the conventions on procedure calls.

Setting Parameters

All parameters are pushed on the stack and are set in the parameter area. The first parameter in the source program is placed at the lowest address. This is illustrated in Figure 7-1.

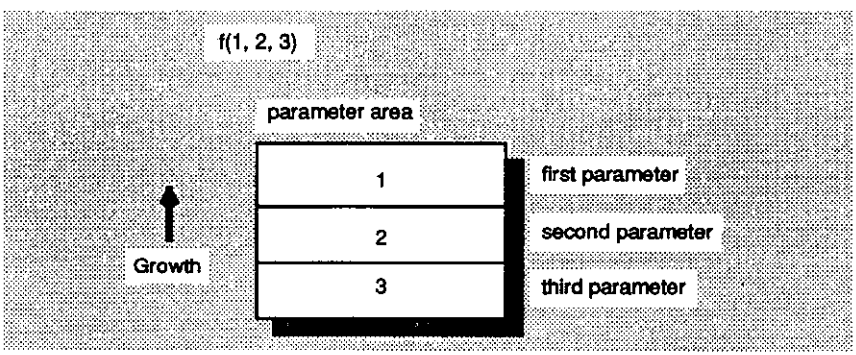

Figure 7-1. Parameter Area of the Stack

Setting Short Integers

Short integers (byte or word) are extended to a long word according to its sign before it is set in the parameter area. This is illustrated in Figure 7-2.

However, if a short argument is passed to a short prototype parameter, then the short integer is not extended to a long word.

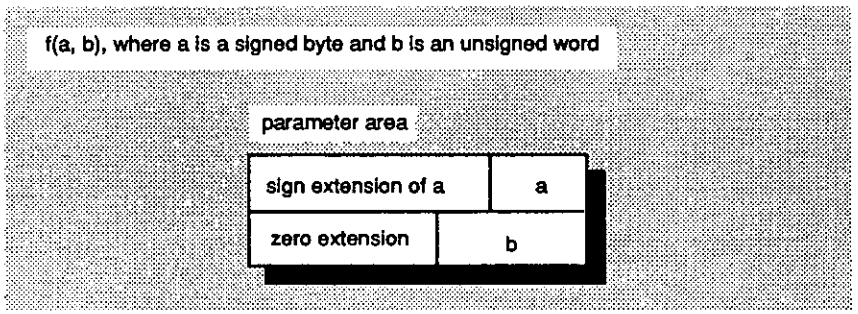

Figure 7-2. Parameter Area and Short Integers

Implicit Parameter for Structure/Union Return Value

When the return value type is a structure or union, the calling function pushes the pointer to the area in which the return value is set by the called function. This return value area should be allocated by the calling function. This is illustrated in Figure 7-3.

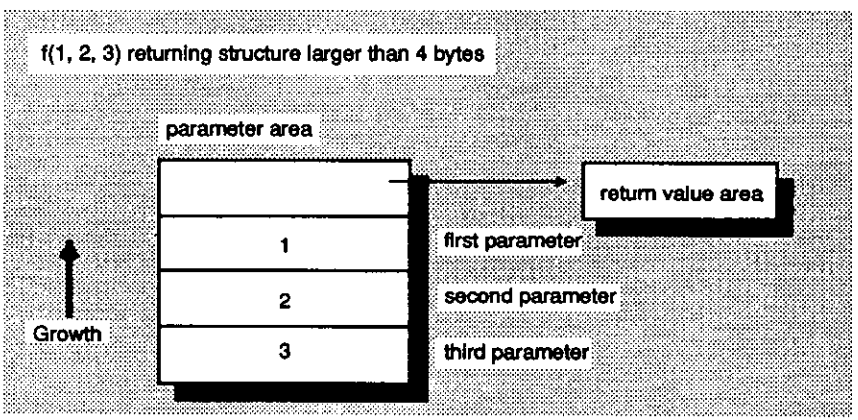

Figure 7-3. Parameter and Return Value Areas

Alignment of Structure/Union in Parameter Area

If the parameter is a structure or union, and its size is not a multiple of 2, the size of the parameter area is rounded up to be a multiple of 2 and the parameter data is set at the beginning of this area. This is illustrated in Figure 7-4.

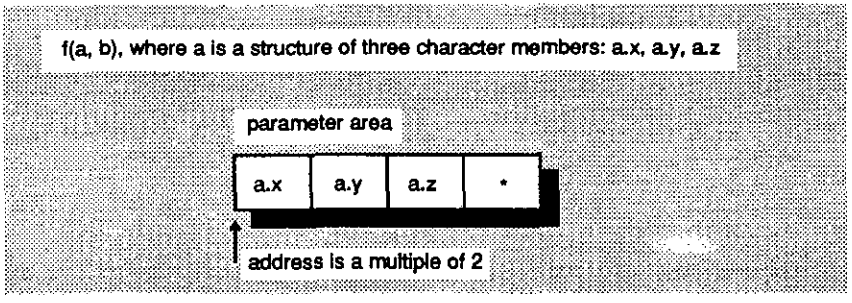

Figure 7-4. Parameter Area and Structure/Union

Function Return Values

This section describes the type of return values that are possible.

Integer Return Values

An integer or pointer type return value is set in **D0**. An integer return value with a size less than 4 bytes may not be extended to a long word before return.

Floating-Point Return Values (with FPU)

A floating-point return value is set in **FP0**.

Floating-Point Return Values (without FPU)

A floating-point return value is returned in **D0/D1**, with **D0** containing the high order value.

Structure/Union Return Value

A structure or union return value is returned through the implicit first parameter which points to the return value area.

When a structure return value is assigned to another structure or pushed onto the stack as a parameter, the implicit parameter which indicates the return value area can be selected as follows:

```
struct s {
    int  r, i;
} a, f();       <implicit parameter is the address of a>
   :
   a=f ();              PEA         _a
      :                 JSR         _f
      :                 ADDQ.L      #4,SP

                   <implicit parameter is the address of
                    parameter area>
   g(f(),1);            PEA         1
                        PEA         -8(A6)
                        JSR         _f
                        ADDQ.L      #4,SP
                        LEA.L       (A6),A0
                        MOVE.L      -(A0),-(SP)
                        MOVE.L      -(A0),-(SP)
                        JSR         _g
```

Popping Parameters

The parameter area is popped by the calling function.

Implicit First Parameter for Structure/Union Return Value

There is an implicit first parameter for a structure or union return value.

The code fragment below shows the caller/calling function interface.

```
int  f (int  a,...)          _f:    LINK        A6,#0
                                    UNLK        A6
                                    RTS
{
}

   :                                PEA         1
   f (1);                           BSR.W       _f
   :                                ADDQ.L      #4,SP
```

Register and Stack Usage with Functions

The Microtec Research C compiler uses register A7 as the stack pointer and register A6 as the frame pointer. All other registers are used to store temporary variables and intermediate results.

In addition, the compiler can be forced to reserve a register (A2 through A6) by using the /reserve (VMS) or -Mr (UNIX/DOS) option. When a register is reserved, the compiler does not generate any code that uses the reserved register. If the A6 register is reserved, A5 will be used as the frame pointer.

Functions that are not declared as interrupt handlers are assumed to destroy the registers D0, D1, A0, and A1 (and FP0 and FP1 if a floating-point coprocessor is present). At the return of a function, the condition codes are undefined. All other

registers that are used will be saved and restored automatically by a called C function. Assembly language routines that are called from C must also save and restore all registers that are used, except for registers D0, D1, A0, and A1 (and also FP0 and FP1 if the floating-point coprocessor is present).

Stack Frames

Local frames for functions are automatically generated by the Microtec Research C compiler if a function requires local stack storage. Frames are generated by a LINK instruction on entry and an UNLK instruction on exit. When a frame is generated, local stack storage and variables are accessed on the stack with the A6-relative addressing mode.

If a function does not require local stack storage, local frames are not generated by default. When a frame is not generated, variables and parameters are accessed on the stack with the A7-relative addressing mode. You can force the use of local frames by using the /frames (VMS) or -Kf (UNIX/DOS) option.

When a function returns, any arguments pushed on the stack are removed. If several function calls follow each other in a basic block, the stack will continue to grow until the end of the block. You can force the compiler to pop the stack after each function call by using the /optimize=nocombine_pops (VMS) or -nOc (UNIX/DOS) option. The following instructions shown in Table 7-1 are used depending on how many bytes need to be removed.

Table 7-1. Instructions for Removing Bytes from Stack

Number of Bytes to Remove	Instruction to be Used
1 to 8	ADDQ.L XX,SP
8 to 32768	LEA X(SP),SP
More than 32768	ADD.L X,SP

Function frames in excess of 32K bytes are considerably less efficient than smaller frames. The 68000 addressing modes are designed for 16-bit offsets. When offsets are larger than 16 bits, every machine instruction with a large offset will have one to three instructions added to it in order to bring the offset within range.

The Prologue

The prologue is the sequence of code at the beginning of a function. The prologue sets up a local stack frame and a frame pointer so that function parameters and automatic variables can be accessed through the pointer. The prologue can also include code that saves certain registers on the stack.

You can write your own assembly routines to interface to C programs. If the routines change the contents of any of the registers D2 through D7, A2 through A5, or FP2 through FP7, you must save the old values.

The following example illustrates the recommended instruction sequence for entry to an assembly language routine.

Example:

```
entry:
        link      a6,#-N
        movem.l   reglist,-(sp)
```

In this example, `N` is the number of words allocated for local variables and `reglist` is the list of registers that the routine can modify.

If this sequence is used, the last argument pushed by the function call (which is also the first argument given in the call's argument list) is at address (A6+8). Subsequent arguments begin at (A6 + 8 + the size of preceding arguments).

Local Variables in the Prologue

Local (automatic) variables are located on the stack with negative displacements from the frame pointer, and there can be a maximum of 2^{31}-1 bytes addressable by the frame pointer with negative displacements.

Function frames larger than 32K bytes are considerably less efficient than smaller frames. The 68000 addressing modes are designed for 16-bit offsets. When offsets are larger than 16 bits, every machine instruction with a large offset will have one to three instructions added to it to bring the offset within range. For more information, see *Stack Frames* above.

If you write your own applications and the assembly language routine does not have function arguments and local variables, it is not necessary to set up a local stack frame. Your routine should save the contents of the registers D2 through D7, A2 through A5, and FP2 through FP7, if they are used.

The Epilogue

The epilogue is the sequence of code at the end of a function. The epilogue restores the saved registers and resets the stack frame to the condition at entry. It then passes control back to the calling routine with an RTS instruction.

Registers that are saved in the prologue should be restored first. The local stack frame can be reset by loading the stack pointer with the frame pointer. The calling routine's frame pointer is then restored by popping the saved value from the stack.

The following is the recommended instruction sequence for exiting a routine:

```
movem.l  -N(a6),reglist
unlk     a6
```

In this example, `N` is based on the number of words allocated for local variables and the number of registers in `reglist`.

Assembly Language Interfacing

When interfacing to an assembly language routine, it is recommended that you create a minimal local stack frame. This provides an easy way of keeping local data on the stack. It will also allow the Microtec Research debugger to display assembly language routine names in the procedure traceback viewport.

To write an assembly language routine that can be called from C, perform these steps:

1. Reference C global and external variables.

   Prefix all global (i.e., public) and external names with an underscore.

2. Create a local stack frame.

   Create a minimal stack frame using the LINK instruction. LINK saves the old frame pointer (A6) of the calling routine on the stack and makes the new frame pointer (A6) point to the old frame pointer.

   The LINK instruction can also create a local data storage area in the stack frame if needed. By creating local stack storage in this way, you will not have to pop off the exact number of arguments that were pushed onto the stack when returning from the assembly language routine.

   Example:

   ```
   link a6,#-8
   ```

3. Save registers.

   Save any of the registers D2 through D7, A2 through A5, and FP2 through FP7, if they were used in the assembly language routine. If you did not create a local stack frame, you must also save the registers A6 and A7 if they were destroyed by the assembly language routine. All registers should be saved in numerical order immediately after the stack frame is built. Data registers should be saved first, followed by address registers, then floating-point registers.

   Example:

```
movem.l  d2/d3/a2/a3,-(sp)
fmovem.x fp2/fp3/fp4,-(sp)
```

4. Reference Arguments (optional).

   You can reference arguments passed to the routine via the frame pointer (A6) or the stack pointer (A7). If you use A7, the offsets to the arguments must allow for the size of the local stack frame plus any saved registers. The size of the stack frame is the size of the A6 register (4 bytes), plus the amount of local storage allocated. If you use A6, offsets to the arguments must allow for the size of the saved A6 value only.

5. Return values.

   Return values should be correctly placed before returning. For more information, see *Function Return Values*.

6. Restore registers.

   At the end of the assembly language routine, the saved registers and the old stack frame must be restored. This must be done immediately before the return instruction.

   Example:

```
fmovem.x -60(a6),fp2/fp3/fp4
movem.l  -24(a6),d2/d3/a2/a3
unlk     a6
rts
```

Assembly Language Routine Example

A prototype of an assembly language routine (named `_foo`) that can be called from a C function is shown below.

```
_foo: link a6,#0          ; Creates stack frame.
                          ; Allocates 0 bytes of space
                          ; for local data. Saves
                          ; original A6. A6 now points
                          ; to saved A6 value on stack.
movem.l d2/d3/a2,-(sp)    ; Pushes 3 registers on stack.
                          ; This instruction is used
                          ; only if registers D2-D7,
                          ; A2-A5 are used in the
                          ; assembly language routine
                          ; body.

                          ; Stack Frame:
                          ; 2nd argument        A6+12   SP+24
                          ; 1st argument        A6+8    SP+20
                          ; A6 -->Original A6   A6+0    SP+12
                          ;       Original A2   A6-4    SP+8
                          ;       Original D3   A6-8    SP+4
                          ; SP -->Original D2   A6-12   SP+0
.
.
                          ; Body of procedure.
.
.
movem.l -12(A6), d2/d3/a2 ; Gets saved registers from the
                          ; stack. Values are not popped.
                          ; The stack pointer is not
                          ; used because other values may
                          ; have been pushed onto the stack
                          ; in the procedure body. This
                          ; uses the A6 register, and the
                          ; constant offset (12) shown
                          ; above. You must compute it by
                          ; adding the value given to the
                          ; LINK instruction (0) and 4 *
                          ; the number of address and data
                          ; registers saved (3);
                          ; 0+(4*3)=12. It is a negative
                          ; offset because it is below the
                          ; A6 register. The UNLK below
                          ; will set the stack pointer back
                          ; to its original value. The
                          ; instruction is used only if
                          ; registers D2-D7 or A2-A5 are
                          ; used in the procedure body.

unlk a6                   ; Undoes the stack frame.
                          ; Restores A6 to original value.

rts                       ; Returns to caller.
```

Variable References from Assembly Routines

The compiler automatically prepends a leading underscore to all variable names. The options described under *Modify Naming Conventions* in the *User's Guide* can be used to change this behavior; however, they are not recommended because they prevent access to the standard C library functions (all library functions have a leading underscore).

Interrupt Handlers

Interrupt handlers perform the necessary functions when an interrupt occurs. They preserve the values in various registers and storage locations and transfer control to routines to service the interrupt.

By default, the Microtec Research C compiler assumes that the values in registers **A0**, **A1**, **D0**, and **D1** may be destroyed by each called function. Therefore, you may want to declare a function as an interrupt handler to force the values of these registers to be saved on the stack on entry to the function and to be restored on exit.

A function can be declared as an interrupt handler by using the **interrupt** keyword. Interrupt functions should have no parameters and should return the **void** type.

Example:

```
interrupt void handler (void)
{
...
}
```

By default, a function declared as an interrupt handler will return to the calling function with the **Return To Exception (RTE)** instruction rather than the standard **RTS** instruction.

If the **/ireturn=subroutine** (VMS) or **-Kr** (UNIX/DOS) option is used, the interrupt handler will return with an **RTS** instruction.

Embedded Environments 8

Embedded Applications

An embedded system consists of a self-contained microprocessor and its associated peripheral hardware. The microprocessor's software provides all operating system functions such as input/output, memory allocation, scheduling, and interrupt handling.

Typically, an embedded system does not have a standard operating system, loader, or disk. Instead, its program code and constants usually reside permanently in Read Only Memory (ROM). The program starts execution at a memory location determined by the hardware whenever a RESET occurs (at power-up).

In-Line Assembly

Embedded applications require close and efficient control of the target architecture. To accommodate this need, the Microtec Research C compiler supports assembler statement intermixing with C code.

You can include any number of well-formed assembler lines by using the pseudo-function asm (which can also be specified as ASM).

The asm pseudo-function behaves from the outside exactly like a function (that is, it takes parameters and returns a type). However, asm does not generate a procedure call. Instead, it inserts your assembler code "in-line".

Syntax and Constraints

Syntax:

```
asm([type,] string{,string});
```

or

```
asm([type]);
```

Description

type — An option type that tells the compiler what is returned and, implicitly, where it is returned by the asm invocation.

string Any number of strings (including zero) that contains the assembler instructions. Each *string* separated by a comma corresponds to a new assembler line. (Always include at least one white space in front of any assembler mnemonic.)

By default, the value returned by the asm pseudo-function is of type int. This can be easily modified by specifying a type with the same syntax you would use for the type operand of sizeof().

Because asm behaves like any C function, the global optimizer takes this into consideration and disables certain optimizations.

Simple Use of asm

This section shows some simple examples of how to use asm to put assembler lines into a C program.

Example:

```
main()
   {
   asm(" move.l $1000,SP ; initialize stack pointer");
...
```

In the example above, the quoted string will be inserted immediately after the prologue code generated by the compiler for `main( )`. Only a newline will be added to avoid concatenating this string with the next line.

There is no limit to the number of assembler lines that can be inserted, with a newline at the end of each line.

Example:

```
asm(" sub #8,SP        ; make room",
    " move.l SP,8(SP)",
    " move.l _errno,4(SP)"          );
```

Returning a Typed Value

Unlike other C compilers, Microtec Research C compiler's asm pseudo-function optionally accepts a type for its first parameter. This type tells the compiler what is returned and, implicitly, where it is returned by the asm invocation.

Example:

```
int *sp=asm(int *, " move.l SP,D0");
```

In the example above, since the compiler knows a pointer is returned in D0, the compiler will, after the pseudo-invocation of asm, generate code to move D0 to the variable sp.

Example:

```
double d=asm(double, " fmove FP1,FP0");
```

In this example, since floating-point values are returned in FP0 (assuming a coprocessor), the value of FP0 will be moved into d.

Note that no real call to asm takes place. The passing of the parameters and the call itself are replaced by the string(s) that you supply. You should store in D0/FP0 (or whatever the calling conventions dictate) the value that interests you.

Using volatile with asm

The use of the volatile attribute is recommended when using machine registers in order to prevent the compiler from assuming that the values are permanent. For instance, the example in the previous section, *Returning a Typed Value*:

```
int *sp=asm(int *, " move.l SP,D0");
```

could have been written as:

```
volatile int *sp=asm(volatile int *, " move.l SP,D0");
```

and:

```
double d=asm(double, " fmove FP1,FP0");
```

could have been written as:

```
volatile double d=asm(volatile double, " fmove FP1,FP0");
```

To make our examples more readable, the volatile keyword has been consistently omitted. However, you should consider using it when appropriate.

Using #define for Readability

You can make asm statements more readable with the #define directive.

Example:

```
#define D4 asm(int, " move.l D4,D0");
#define D5 asm(int, " move.l D5,D0");

if (D4>D5)
...
```

This example shows that it is possible to access every single machine device at the C level. The overhead is confined to a move. For D0 (and FP0), not even this move is required.

Variable Names Inside asm

You can specify global and local variables by name inside an asm string. A global variable x can easily be represented at assembly level by _x. Local variables, on the other hand, are represented by frame offsets or by registers that can change from one release of the compiler to the next. Their addresses can even change between two compiles if new code or variables have been added to a procedure. Hence, there is no easy way to represent local variables.

Back Quoting Variable Names

The Microtec Research C compiler lets you insert variable names in any asm string by simply quoting them with a back quote (`). Since it is unlikely that this character will appear in any assembler statement, Microtec Research has chosen it as the default.

Example:

```
foo()
{
int automobile, garage;

asm(" move.l `automobile`, `garage`");
...
```

The actual addresses of the inserted variables will be supplied while conforming to the usual scope rules. According to the memory/register binding in force for the compilation, the above example will generate:

```
move.l 12(a6),d4
```

If you want to use a character other than the back quote, use the /insert_char=*char* (VMS) or -ui*char* (UNIX/DOS) option to change the insert character to *char*.

It is recommended that you use the -ui*char* option inside a #pragma option instead of on the invocation line because you may not want to change the insert character from compile to compile.

Inserts and Global Variables

Inserts are still recommended for global variables. Variable names inside a string are not affected by compiler options like /prepend=dot (VMS) or -upd (UNIX/DOS). However, inserts are option-sensitive.

Example:

```
#pragma option -upd
/* prepend a dot to all global names */
...
asm("·move.l _global,_global2");
```

In the example above, the -upd option cannot affect the global names inside the string.

```
asm(" move.l `global`,`global2`");
```

In this example, the inserts will reflect the -upd option.

asm Versus #pragma asm/#pragma endasm

Since asm is a pseudo-function, it can only be used where a normal function can be invoked, namely inside a procedure. Outside a procedure, any number of unquoted assembler instructions can be inserted between the pair #pragma asm and #pragma endasm.

Example:

```
#pragma asm
#if _FPU
   fmove.l      ROUNDING,FPCR ; set rounding mode
#endif
#if _CHAR_SIGNED
 #if (_68020 || _68030 || _68040 || _CPU32)
   extb.l       D1
   move.l       D1,D0
 #else
   ext.w        D1            ; sign extend
   ext.l        D1
   move.l       D1,D0
   #define XXX signed.s
 #endif
#else
   move.q       #0,D0         ; zero extend
   move.b       D1,D0
   #define XXX unsigned.s
#endif
#include XXX
#pragma endasm
```

Note the use of user-defined macros such as XXX and a predefined one such as _CHAR_SIGNED. The full preprocessor functionality is available inside the pair #pragma asm/#pragma endasm. However, inserts cannot be used.

Other Examples

Example:

```
unsigned char save_D1=asm(unsigned char, " MOVE.L D1,D0" );
```

This example shows how to store the contents of register D1 in the user variable save_D1.

The compiler produces the following sequence:

```
MOVE.L D1,D0
MOVE.B D0,_save_D1
```

Example:

```
asm( " MOVE.B $FE,4(A6)" );   /* move the contents */
/* of an absolute location to a position on the stack */

asm(" CMP $80,_mask"," BEQ label", " TRAP #0","label: ");
/* conditionally raise an interrupt */
```

This example shows how to use asm to move the contents of an absolute location to a position on the stack and to then raise an interrupt.

As for any other C function, the default and ignored return value are of type int.

Valid alternatives for the above statements are:

```
asm(void," MOVE.B $FE,4(A6)" );
(void) asm (" CMP $80,_mask"," BEQ label"," TRAP #0","label: ");
```

Example:

```
carry_set=asm(" SUBX D0,D0; 1 if carry set ");
```

This example determines if the carry flag is set or reset by subtracting D0 and the carry from D0 itself. If the carry is reset, the subtraction will place zero (false) into D0; otherwise, D0 will contain non-zero (true).

The compiler produces this sequence in the assembly file:

```
SUBX D0,D0; 1 if carry set
MOVE.L D0,_carry_set
```

Example:

```
pi=asm(double," FMOVE.X FP4,FP0");
```

This example shows how to fetch the value of a floating-point register FP4. Notice that real values are generally returned in the floating-point register FP0.

Examples:

```
x=asm(double," fmove FP1,FP0") * asm(double," fmove FP2,FP0");

printf("sin(x)=%g",asm(double," fmove 'x',FP0"," fsin FP0"));

if (asm(int, " move.l D4,D0")>=asm(int, " move.l D2,D0")) ...

int d0=asm(); /* returns D0 */

if (asm(boolean, " move.l CCR,D0") & 0x0030) overflow();
```

```
sp_before = asm(volatile int *," move.l SP,D0");
      Denmark();

sp_after = asm(volatile int *, " move.l SP,D0");

if (sp_before != sp_after)
      printf("There is something rotten in Denmark");
```

In the examples above, floating-point values are returned in `FP0`.

Examples:

```
asm( " NOP"," TRAP #0");

asm( " NOP\n TRAP #0" );

asm( " NOP "," T" "RAP #0 " );
/* no ','after T - same as " TRAP #0 " */

asm( " NOP "); asm( " TRAP #0 " );

asm(int," NOP"," TRAP #0");
```

All of these statements generate:

```
NOP
TRAP #0
```

Example:

```
asm(char*, " MOVEA.L #0,A0 " ) [100]='*';
```

This example shows how to impose a `char` array view on physical memory.

Example:

```
if ( asm(struct {void *ret_addr; int parm1, parm2;}*,
          " MOVEA.L A6,A0 ") -> ret_addr<0x8FFFFFFF) ...;
```

This example lets you inspect the current procedure activation record.

Examples:

```
asm("label:"); /*label is in col 1 */

asm("MOVE.L #1,4(A6,D2.B); /* ERROR - MOVE in col 1 */

asm(" ORG.S $100");

asm("thous DC.L $1000,$2000,$3000 " );

asm(" garbage in ...");
```

These examples show how powerful and potentially dangerous the asm feature can be. Use the asm pseudo-function with caution.

Guidelines for Assembler Inlining

The following guidelines for assembler inlining should be used:

- Use #pragma asm/#pragma endasm only outside procedures. Use asm() inside procedures.
- Use inserts only in asm() strings.
- When returning machine register contents, use volatile.

Considerations for Assembler Inlining

The content of any string is passed to the assembler as-is. No control is placed on it whatsoever. Direct consequences of this are:

- You must be ready to read errors issued by the assembler.
- The compiler makes pessimistic assumptions on the size of the inserted instructions. Every jump crossing the instructions might be long. Consider the following example:

```
asm(" DS $1000")
```

- asm is not portable. It is likely, however, that a program which uses direct machine instructions was never meant to be portable.

  Example:

```
printf( "%x",asm() );
```

  This prints the contents of register D0. The data printed could be completely different if the program is compiled on another compiler.

  When you use asm, check the effects of optimizations. Even though some optimizations are disabled, the Microtec Research C compiler takes quite a bit of freedom in rearranging, changing, and deleting code. This could affect your program.

Considerations for Embedded Systems

You should consider the following when working with an embedded system:

1. The hardware starting address must be made to coincide with the main entry point (or code that ultimately jumps to the main entry point). For this reason, the example initialization routine in the file entry.c might need to be modified. If any modifications are made, entry.c must be recompiled, and the resulting object module must be linked with your program. The initialization routine is described in detail later in this chapter.

2. Initialized variables can be placed either in ROM or RAM, or initialized by the program at runtime. If variables are placed in ROM, they cannot be altered by the program.

3. To use the C I/O and memory allocation library, several routines will have to be modified to adapt them to your embedded environment. For more information, see *User-Modified Routines* in a later section of this chapter.

Removing Unneeded Floating-Point Support

If **printf**, **fprintf**, or **sprintf** is used and no floating-point values are ever read, you can avoid linking in the full floating-point support by adding the following stub for _ftoa and compiling and linking it before you link in the library:

```
int _ftoa() { return 0; }
```

Similarly, if **scanf**, **fscanf**, or **sscanf** is used and no floating-point values are ever read, you can avoid linking in the full floating-point support by adding the following stub for _sfef and compiling and linking it before you link in the library:

```
void _sfef() { return; }
```

In both cases, this will reduce the size of the executable code.

System Functions

In addition to the functions provided in the Microtec Research C run-time library, the compiler assumes the existence of UNIX-style system functions. The source code to these system functions are provided with the libraries.

You must also modify the functions **sbrk** and **_exit** as well as the routines **inchrw**, **outchr**, **_START**, and **entry**. Descriptions of these routines are given in the section titled *User-Modified Routines*. The system functions are described in Chapter 5.

Sample versions of the system functions **inchrw**, **outchr**, **_START**, and **entry** are provided in the libraries.

HP 64000 Systems

The Microtec Research C compiler provides support for the HP 64000 Series Development System. This support allows you to download a software application with an HP-supplied transfer program, and then symbolically debug it on an HP 64000 system.

HP 64000-compatible files are generated through the use of the **hp64000** command line option **/hp64000** (VMS) or **-h** (UNIX/DOS). When HP support is used, the output file names are generated as described in Table 8-1.

Table 8-1. HP Output File Names

Suffix	Produced By	File Type
.s	Compiler	Assembly source code
.A	Assembler	Assembler symbol file (asmb_sym)
.X	Linker	Absolute object module file
.L	Linker	Linker symbol file (link_sym)

The files listed above can be downloaded to the HP 64000 via a host support transfer utility available from HP. Once these files have been downloaded, they can be symbolically debugged using the HP 64243 emulator, the HP 64620 logic state/ software analyzer, and the HP 64310 software performance analyzer modules.

High-level language support for simulated I/O on the HP 64000 is also included with the product distribution.

Using HP 64000 with the Microtec Research Compiler

These considerations should be made when using the Microtec Research C compiler for development in the HP 64000 environment:

- The HP 64000 development system allows only three relocatable sections per module. The options described under *Name the Sections* in the *User's Guide* should be used to create the section names shown in Table 8-2.

Table 8-2. Section Names for HP 64000 Development Systems

Section	Default Name	Change To:
user const variable	const	13
uninitialized data	zerovars	14
initialized data	vars	14
compiler literals	literals	13
strings	strings	13
code	code	9

In addition, the user-supplied routine for dynamic memory allocation, **HPSBRK**, should be modified to allocate heap from either the **DATA** section or the **COMMON** section.

- The line number symbols generated by the compiler are converted to the form *#nn* when the assembler produces the assembler symbol file (asmb_sym).

  Avoid creating symbol names that have the form *#nn* since this form is used for line number symbols in the assembler symbol file. User symbols in this form will cause "duplicate symbol" errors.

- Avoid creating symbol names that include the characters ?, ., and $ since these characters are translated into an underscore (_) by the assembler and linker.

- When the **/hp64000** (VMS) or **-h** (UNIX/DOS) option is used, two special labels, R*label* and E*label*, are generated by the compiler for each function. The HP 64000 uses these labels to identify each routine's return point and end address for debugging and timing purposes. The R*label* symbol represents the routine's return point, and the E*label* symbol represents the routine's end address. The *label* portion of the symbol name is the name of the routine.

Using HP 64000 with ASM68K

When using the ASM68K Assembler and LNK68K Linker for HP 64000 development support, consider the following:

- Symbol names are limited to 15 characters in length. Characters specified beyond the 15 character limit are ignored. Symbols that are non-unique in the first 15 characters are flagged with duplicate symbol errors at assembly time.

- Symbol names cannot include any question marks (?), periods (.), or dollar signs ($). If a symbol name contains one of these symbolic characters, the character is automatically converted into an underscore (_). This conversion can result in duplicate user symbols, in which case they will be flagged as duplicate symbol errors at assembly time.

- Sections are limited to the four predefined sections supported by the HP 64000. These sections are absolute, PROG, DATA, and COMN. ASM68K accepts both the HP and Microtec Research directives listed in Table 8-3.

Table 8-3. Microtec Research and HP Section Directives

Microtec Research	HP
ORG	Absolute
SECTION 9	PROG
SECTION 14	DATA
SECTION 13	COMN

Any sections other than those listed above are flagged as incompatible with the HP environment at assembly time and are ignored.

HP 64000 Requirements

When using the HP 64000 system, you must adhere to the following requirements:

- Object modules loaded by the LNK68K Linker must have the same module name as the corresponding source file if local symbols and source lines are to be referenced by the HP 64000 modules.

- To communicate with the emulator on the HP 64000, you need to include the 68K monitor module supplied with the HP 64000 system in your absolute object module. This is done following the procedure listed below:

  1. Upload the 68000 monitor module to your host system.

  2. Make minor modifications to the source code to correct syntax incompatibilities with Microtec Research and Motorola formats. These modifications are summarized in Table 8-4.

Table 8-4. Modifications For Syntax Incompatibilities

Microtec Research		HP	
TTL	'68000'	"68000"	
XDEF	SYMBOL	GLB SYMBOL	
PAGE		SKIP	
ORG	$0BC	ORG 0BCH	
CMP.L	#MONITOR_START,2(A7)	CMP.L	#MONITOR_START,2[A7]
MOVE.L	(SP)+,PSTATUS	MOVE.L	[SP]+,PSTATUS
OR	$F8FF,D0	OR	0F8FFH,D0
DC.B	'Error! Entry Mode=User'	ASC	"Error! Entry Mode=User"
DC.W	REG_END-*	DC.W	REG_END-$"
SECTION	PROG	PROG	
BNE.S	MOVE_WORD	BNE.B	MOVE_WORD

3. Assemble the modified source code using the ASM68K Assembler.
4. Include the assembled source code file name in your linker option file. The linker option file should specify all of the other modules that need to be linked in to your absolute object module.
5. Link all of the object modules using the LNK68K Linker.

User-Modified Routines

This section describes the routines that you must modify when using the Microtec Research C compiler in certain environments. Sample versions of these routines are provided on the distribution, but it is assumed that these sample routines will be modified extensively or used only as examples for writing your own routines.

The first part of this section covers the routines needed for users running on either an embedded system or a non-UNIX system. The second part covers the routines needed for users running on an HP 64000 system.

User-Modified Routines for Embedded Systems

If you are running on a non-UNIX or embedded system, the routines **inchrw**, **outchr**, **entry**, **_START**, and the functions **sbrk** and **_exit** must be modified to work with your embedded system environment. These routines are described below. Simple versions of these routines and functions are provided in the files **INCHRW.C**, **OUTCHR.C**, **CSYS.C**, **ENTRY.C**, and **S_SBRK.C**. These sample routines will work with XRAY and are sufficient for initial debugging in that environment.

1. In Character Routine (inchrw() in INCHRW.C)

   The inchrw() routine reads characters from the _simulated_input variable. The XRAY INPORT command may be used to specify the source of input data for _simulated_input:

   ```
   inport &_simulated_input [,input_source]
   ```

   If no INPORT command is issued, XRAY will read from the standard I/O screen.

   inchrw() returns an integer with a value between 0 and 255, or -1 if an error occurs.

2. Out Character Routine (outchr() in OUTCHR.C)

   The outchr(ch) routine writes characters to the _simulated_output variable. The XRAY OUTPORT command may be used to specify the destination of the output data for _simulated_output:

   ```
   outport &_simulated_output [,output_destination]
   ```

   If no OUTPORT command is issued, XRAY will write to the standard I/O screen. Its return value is ignored.

3. Start Routine (_START() in CSYS.C)

   A simple version of the start routine is included in the file CSYS.C. This version of start initializes the I/O system and opens the standard files stdin, stdout, stderr, stdaux, and stdprn. It then calls your main() function with the three arguments argc, argv, and environ. If or when your main() function returns to start, it will call exit(). The exit function will flush and close all opened files and then call _exit to terminate the program.

   It can optionally call the initcopy routine to copy initialization values from ROM to RAM (see *Using the Linker* in a later section of this chapter for a description of the LNK68K INITDATA command).

   The variables _simulated_input and _simulated_output (used by inchrw and outchr) are declared in CSYS.C. They are placed in section ioports. This section should be located in RAM.

4. Exit Routine (_exit() in ENTRY.C)

   A simple version of the _exit routine is included in the file ENTRY.C. The exit routine is implicitly called at the end of each main() function to return

control to the operating system. In this implementation, exit closes all opened files and then calls _exit.

5. Initialization Routine (entry() in ENTRY.C)

   The initialization routine initializes the embedded system environment. It should meet the following requirements:

   a. It must be written in assembly language.

   b. It should set up values for the stack pointer.

   c. It should provide a static variable for the heap pointer.

   d. It should call the _START routine.

   The sample initialization routine (entry) is the main entry point. It does the following:

   - Initializes SP as .STARTOF.(STACK) + .SIZEOF.(STACK), which will be defined at link time
   - Initializes __HEAP as the starting address of the heap section, which is located after all other sections
   - Initializes the frame pointer
   - Clears the zerovars section
   - (optional, done in mcc68ka5*xx*.lib) Sets the a5 register for a5-relative data addressing
   - Calls the C initialization routine START

   A sample C initialization routine (START) is also included in the library. It does the following:

   - Initializes library global variables
   - Opens the standard files stdin, stdout, stderr, stdprn, and stdaux to the console
   - (optional) Calls the initcopy routine for copying initialized data in the vars section from ROM to RAM
   - Calls your main function
   - If or when your main function returns, it will call exit

   The source code for the sample C initialization routine is located in the file csys.c.

6. Heap Management Routine (sbrk() in S_SBRK.C)

   The sbrk function keeps track of the heap's growth and allocates space from the heap. It increments the heap pointer (__HEAP) by *size* bytes and returns the previous heap pointer value in register D0. If there is not enough space on the heap, sbrk returns -1.

User-Modified Routines for HP Systems

If you are running on an HP 64000 development system, the modules hpentry.src, hpsbrk.src, hpoutchr.c, and hpinchrw.c are provided. These routines are described below.

1. Initialization Routine (entry() in HPENTRY.S)

   The HP 64000 entry routine initializes the stack pointer and calls the appropriate high-level language initialization routine.

2. Heap Management Routine (sbrk() in HPSBRK.S)

   The HP 64000 sbrk routine provides a dynamic memory allocation heap for use in the high-level language environment. It is functionally equivalent to the sbrk function used with embedded systems, but it is internally different. In the HP 64000 environment, this routine defines the initial address of the heap to start at the end of the routine. For this reason, the data section of the HP 64000 sbrk routine must be at the end of user-defined memory.

3. In Character Routine (inchrw() in HPINCHRW.C)

   The HP 64000 inchrw() routine allows characters to be read from the HP 64000 keyboard. Since simulated I/O on the HP 64000 is provided in block transfer mode, the HP 64000 waits to read the characters from the keyboard until either the input buffer is full (80 characters) or until a carriage return character (CR) is detected.

   This routine requires that you specify the PRT__CNTL__0 and DSP__CNTL__0 public symbols at link time to the proper control block addresses ($301 and $501).

4. Out Character Routine (outchr() in HPOUTCHR.C)

   The HP 64000 outchr(ch) routine allows characters to be written to the HP 64000 console. Since simulated I/O on the HP 64000 is provided in block transfer mode, the HP 64000 waits to write the characters to the screen until either the output buffer is full (80 characters) or until a carriage return character (CR) is detected.

This routine requires that you specify the **PRT__CNTL__0** and **DSP__CNTL__0** public symbols at link time to the proper control block addresses ($301 and $501).

Using the Linker

This section provides linker command examples, including an example using a ROM-based system and an example using an HP 64000 development system.

Default Sections

By default, the compiler will use the section names shown in Table 8-5. Alternate section names can be indicated by using the options described under *Name the Sections* in the *User's Guide*.

Table 8-5. Section Names

Section	Default Name	Section Type
code	code	CODE
uninitialized data	zerovars	DATA
initialized data	vars	DATA
user const variables	const	ROM
strings	strings	ROM
compiler-generated literals	literals	ROM
compiler-generated tags	tags	DATA

See an earlier section titled *HP 64000 Systems* for information on section names in the HP 64000 environment. See *Choose Address Mode for Sections* in the *User's Guide* for information on locating and addressing the various code and data sections.

Libraries

The Microtec Research C compiler package includes six libraries that are referred to collectively as MCC68K*xx*.LIB. You should use them as follows:

- The ab libraries should be used when the compiler is generating absolute (position-dependent) code and data references.
- The a5 libraries should be used when the compiler is generating PC-relative code references and a5-relative (position-independent) data references.

- The pc libraries should be used when the compiler is generating PC-relative (position-independent) code and data references.
- The 020 libraries should be used when the compiler is generating 68020/030/040/CPU32 code.

The libraries shown below are built when certain options are set that produce the following kinds of code and data:

mcc68kab.lib	Position-dependent code and data, 68000/010 instructions
mcc68kpc.lib	PC-relative code and data, 68000/010 instructions
mcc68ka5.lib	PC-relative code, A5-relative data, 68000/010 instructions
mcc68kab020.lib	Position-dependent code and data, 68020/030/040/CPU32 instructions
mcc68kpc020.lib	PC-relative code and data, 32-bit displacements, 68020/030/040/CPU32 instructions
mcc68ka5020.lib	PC-relative code, A5-relative data, 32-bit displacements, 68020/030/040/CPU32 instructions

For more information about the ab, a5, pc, and 020 libraries, see the *Generate Position-Independent Code and Data* and *Produce Code for Specified Processor* sections in the *User's Guide*.

Since the compiler generates absolute data references by default, the absolute library, mcc68kab.lib, should generally be used. Refer to the *Produce Code for Specified Processor* section of the *User's Guide* for more information about position-independent data options.

INITDATA Command (IEEE Only)

For information on INITDATA for systems which generate COFF, see the *ASM68K Reference Manual.*

The LNK68K linker command INITDATA provides a method for initializing program variables in RAM at program start-up time. If the INITDATA command is used, the linker will create the section ??INITDATA. This section will contain the initialization values for the section(s) specified with the INITDATA command. At program start-up time, the initcopy function will copy the initialization values from the ??INITDATA section to the section(s) specified with the INITDATA command.

The Microtec Research run-time libraries contain the **initcopy** function. If the LNK68K command is used, the **initcopy** function should be called from the START function in CSYS.C. The CSYS.C file on the distribution includes a call to **initcopy** which is commented out. If any modifications are made, CSYS.C must be recompiled and the resulting object module must be linked with your program.

The **??INITDATA** section should be allocated in ROM. It may be ordered and assigned an address using standard linker commands.

Linker Command Example (IEEE Only)

For linker command examples for systems which generate COFF, see the *ASM68K Reference Manual.*

The following linker command file creates the memory configuration given in Figure 8-1.

```
listabs publics, internals
listmap publics
format ieee                          ; Output file format
extern ENTRY                         ; Force load of initialization
                                     ; routine
sectsize heap=$8000                  ; Set size of heap
sectsize stack=$1000                 ; Set size of stack
common stack=$F000                   ; Set start of stack section
;
order stack                          ; Stack section
order literals,strings,const,code    ; ROM Section
order vars,zerovars,ioports,heap     ; RAM Section
load sieve                           ; User application routine
load mcc68kab.lib                    ; C run-time library
end                                  ; End of command file
```

The compiler places code and data in the sections described previously in *Default Sections*. The example assembly module **ENTRY** uses section **heap** for the heap.

By default, the linker begins assigning memory at address 0. This is unsuitable for many real applications not only because addresses 0 to 3FF are used as interrupt vectors, but also because space needs to be allocated for the stack. This can be resolved by using the **COMMON** command to specify the starting point address of the stack section.

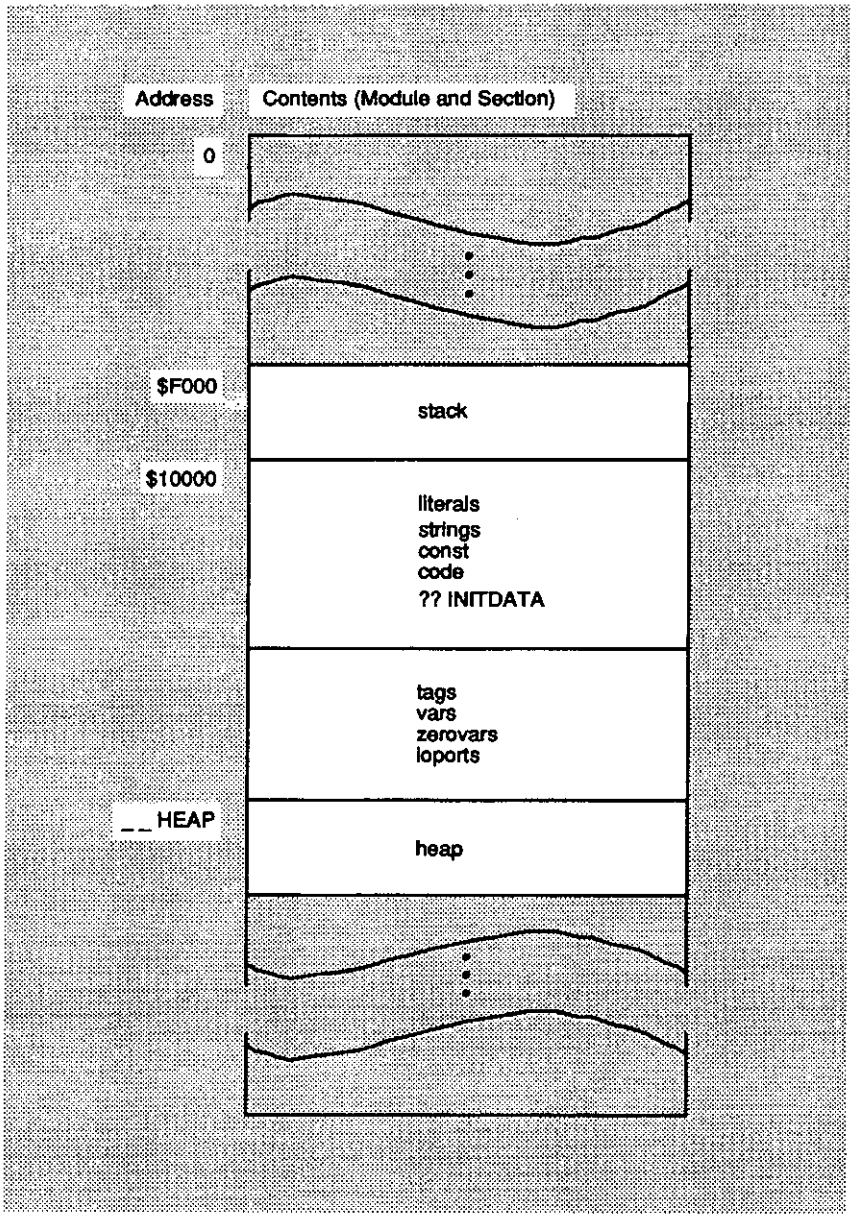

Figure 8-1. Memory Configuration

Linker Command Example (ROM-Based System, IEEE Only)

For linker command examples for systems which generate COFF, see the *ASM68K Reference Manual.*

In some applications, such as ROM-based dedicated systems, code, data, and stack sections are to be assigned to preallocated areas of the processor's address space. In the following linker command file, it is assumed that the literals section is to be placed in the ROM area starting at F000, the zerovars section is to start at location 2000, the vars section is to start at location A000, and the stack is to reside in the 0 to 2000 region.

```
listabs publics, internals
listmap publics              ; Linker options
format ieee                  ; Output file format
extern ENTRY                 ; Force load of initialization routine
sect ??INITDATA=$E000
sect literals=$F000          ; Code section address
sect zerovars=$2000          ; Uninitialized data section address
sect vars=$A000              ; Initialized data section address
sectsize heap=$8000          ; Set size of heap
sectsize stack=$1000         ; Set size of stack
common stack=$F000           ; Set start of stack section
;
order stack                  ; Stack section
order literals,strings,const,code
order zerovars,ioports,heap
initdata vars                ; Put init values in ??INITDATA
load entry                   ; Main entry point
load sieve                   ; User-application routine
load mcc68kab.lib            ; C run-time library
end                          ; End of command file
```

Figure 8-2 shows this memory configuration.

The initialized variables will be allocated in section vars and their initialization values will be allocated in the ??INITDATA section, which will be placed at E000. To copy the initialization values from ??INITDATA (ROM) to vars (RAM), initcopy should be called from the entry function.

Using the allocation map generated by the linker, further adjustments to the starting addresses might be necessary to ensure sufficient space for each program section.

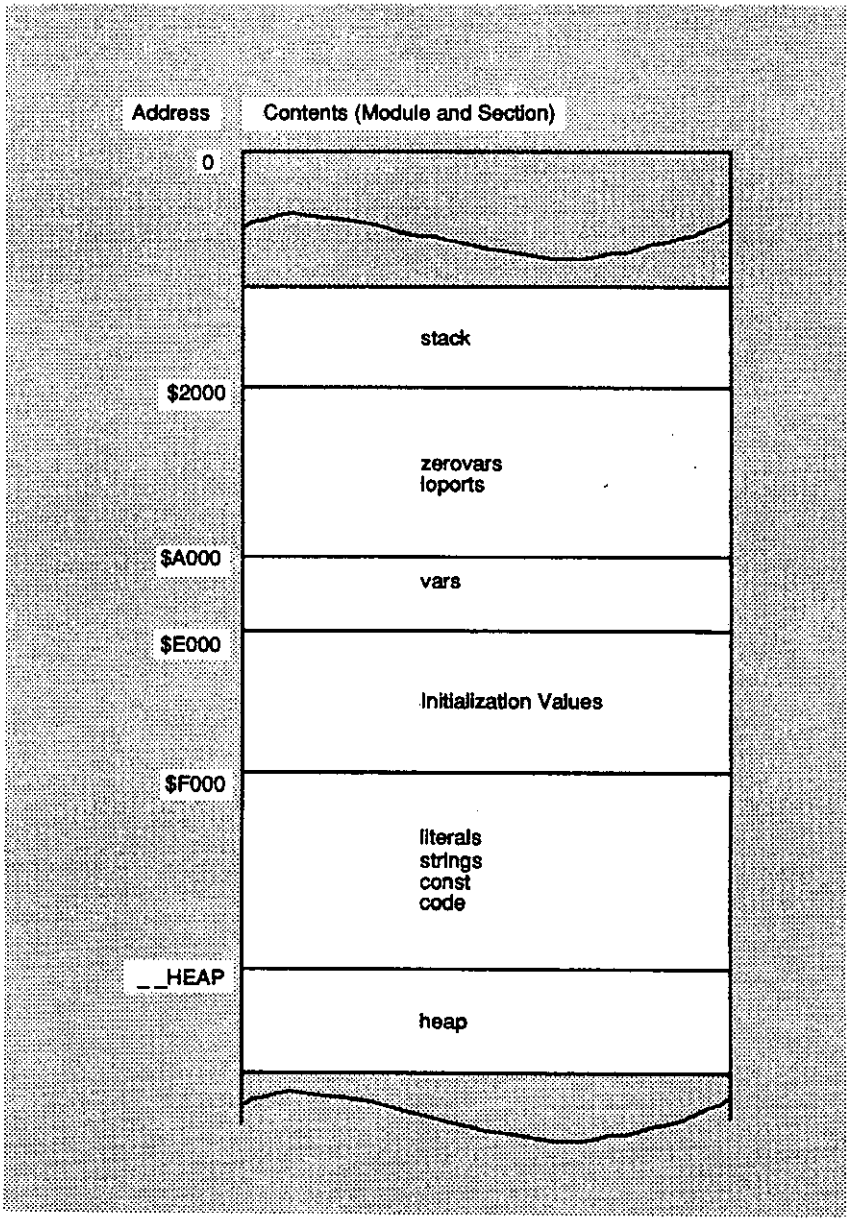

Figure 8-2. Memory Configuration (ROM Based)

Linker Command Example (HP 64000 System, IEEE Only)

For linker command examples for systems which generate COFF, see the *ASM68K Reference Manual*.

When using an HP 64000 development system, code, data, and common sections are to be assigned to preallocated areas of the processor's address space. In the following linker command file, it is assumed that the code section is to be placed in the ROM area starting at 6000, the data section is to start at location B000, and the common section is to start at location 2000. See the section *HP 64000 Systems* for information on section names in the HP 64000 environment.

```
listabs publics, internals
listmap publics
sect      9=$6000                  ; CODE address
sect      13=$2000                 ; COMN address
sect      14=$b000                 ; DATA address
sect      prog=$1000               ; MONA68000
public ????stacktop=$1000          ; Initializes stack pointer
                                   ; (SP)
public _dsc_cntl_1=$802            ; Initializes 64000 I/O
                                   ; Control block
public _dsc_type=$802
public _dsc_cntl_0=$801
public _disc_no=$803
public _no_words=$802
public _record_number=$802
public _prt_cntl_0=$501
public _prt_cntl_1=$502
public _dsp_cntl_0=$301
public _dsp_cntl_1=$302
load      hpentry                  ; Main entry point
load      sieve                    ; User-application routine
load      hpoutchr                 ; Output character
load      hpinchrw                 ; Input character
load      mona68000                ; HP 64000 Monitor module
load      mcc68kab.lib             ; C run-time library
load      hpsbrk                   ; Heap management routine
end                                ; End of command file
```

In the memory configuration shown in Figure 8-3, it is assumed that **SIEVE**, **HPINCHR**, and **HPOUTCHR** have been compiled. The same routines, as well as **HPSBRK** and **HPENTRY**, must also have been assembled using the ASM68K Assembler. It is also assumed that the HP-supplied monitor has been uploaded, converted, and assembled. When using a host other than HP-UX, all compilations and assemblies must be performed using the **/hp64000** (VMS) or **-h** (UNIX/DOS) command line option. See the section titled *HP 64000 Systems*.

The HP 64000 sbrk routine defines the initial address of the heap to start at the end of the routine. For this reason, the data section of the sbrk routine must be at the end of user-defined memory. This is accomplished by loading the sbrk module after the run-time library.

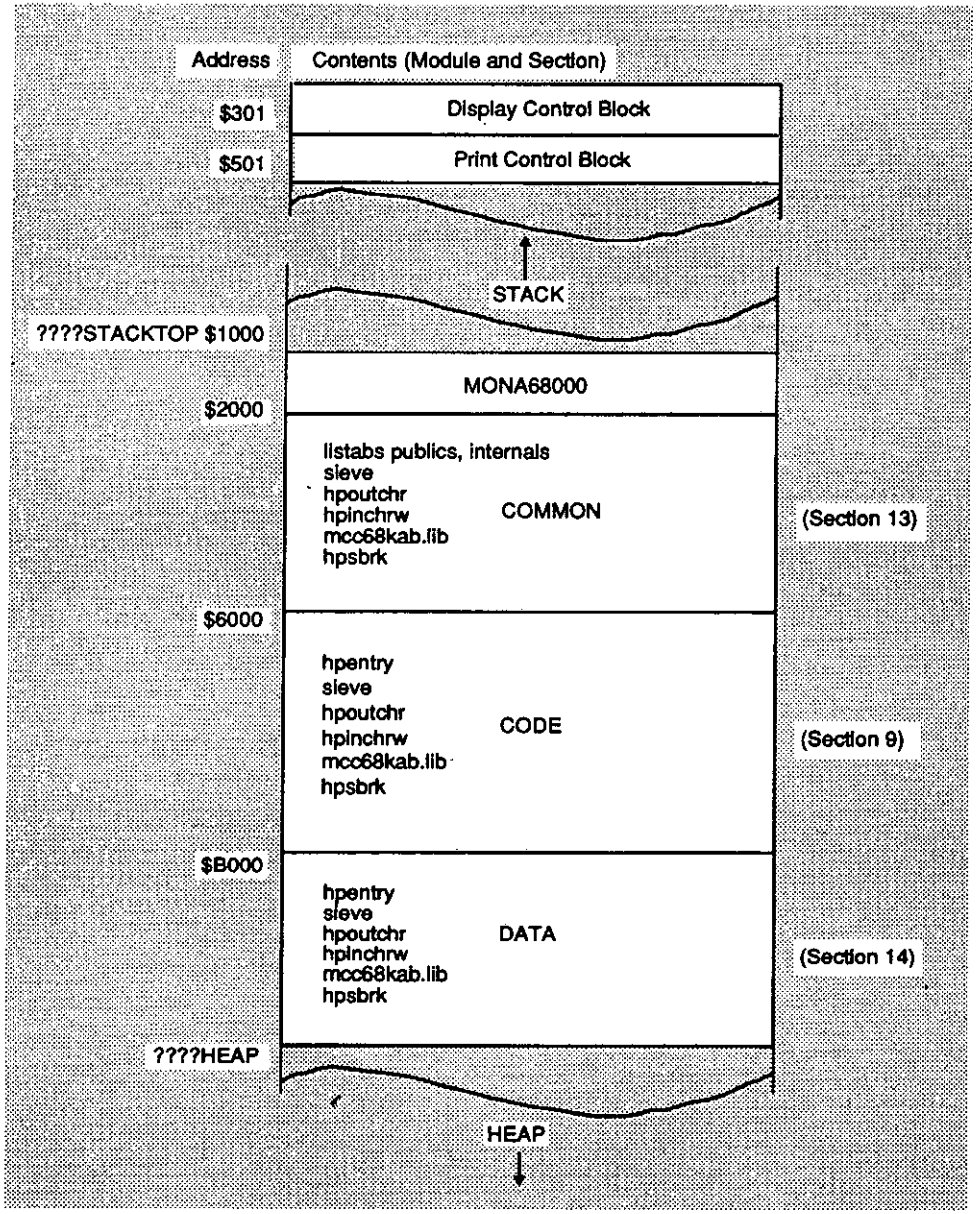

Figure 8-3. Memory Configuration (HP 64000 Based)

Position-Independent Code

This section describes position-independent code, position-independent data, and how they relate to the Microtec Research C compiler, the C programming language, and the 68000 microprocessor.

By default, the compiler generates references to absolute code and absolute data. However, the command line options /code=pc (VMS) and -Mcp (UNIX/DOS), /data=pc (VMS) and -Mdp (UNIX/DOS), and /data=A*n* (VMS) and -Md*n* (UNIX/DOS) let the compiler generate references to position-independent code and position-independent data. These options are described in Table 8-6.

Table 8-6. Position-Independent Code Options

Option	Meaning
code=pc	Generates position-independent code by using program counter (PC) relative addressing for all references to code
data=pc	Generates position-independent data by using program counter (PC) relative addressing for all references to global data
data=A*n*	Generates position-independent data by using A*n*-relative addressing for all references to global data (A*n* is A2, A3, A4, A5, or A6)

For more information about command line options supported by the compiler, refer to the *User's Guide*.

Position-Independent Versus Position-Dependent Code

Position-independent code and data references are used for multitasking systems and for other programs that need to be loaded at different addresses for different executions. A program that is position independent can be loaded by the operating system at any available memory location and will work regardless of where it is placed in memory. However, a stand-alone program targeted for a specific CPU board and memory configuration does not need to be position independent because it is always loaded at the same memory location.

A position-independent program specifies its code and data addresses relative to a register by using register-relative addressing. The /data=pc (VMS) or -Mdp (UNIX/DOS) option uses the program counter as the base register. The /data=A*n*

(VMS) or -Md*n* (UNIX/DOS) option uses A2, A3, A4, A5, or A6 as the base register.

The effective address is generated by adding a displacement (or offset) to the base. The base displacement is a two- or four-byte value that immediately follows the operation code in the instruction. This procedure for generating the effective address works because the relative positions of instructions and data in a program remain the same even though their absolute addresses can change each time the program is loaded.

Compiler Considerations

1. The compiler adheres to the following rules when generating position-independent code references (/code=pc on VMS and -Mcp on UNIX/DOS):
   - The compiler does not use immediate addressing to load address constants.
   - To load the address of a function, the compiler uses the PC-relative form of the Load Effective Address (LEA) instruction.
   - To push the address of a function, the compiler uses the PC-relative form of the Push Effective Address (PEA) instruction.
   - The compiler uses PC-relative addressing for all jumps and internal subroutine or function calls.
2. The compiler adheres to the following rules when generating position-independent data references (/data=pc on VMS and -Mdp on UNIX/DOS, and /data=A*n* on VMS or -Md*n* on UNIX/DOS):
   - The compiler does not use immediate addressing to load address constants.
   - To load the address of a global data item, the compiler uses the register-relative form of the Load Effective Address (LEA) instruction.
   - To push the address of a global data item, the compiler uses the register-relative form of the Push Effective Address (PEA) instruction.
   - The compiler uses register-relative addressing for all references to static or global variables.

Note

Local automatic variables are already position independent because they are located on the stack and are addressed relative to the stack pointer.

Programmer Considerations

To generate-position independent code, you must adhere to the following rules:

- Do not call or jump to absolute code addresses within the program.
- Do not use pointers initialized to absolute code addresses within the program.
- Do not locate a call and its target too far away from each other. The target of a call must be in the range of -32768 to 32767 bytes or -2147483648 to 2147483647 bytes if you use the /long (VMS) or -Ml (UNIX/DOS) option and /cpu=68020, /cpu=68030, /cpu=68040, /cpu=CPU32 (VMS), or -p68020 , -p68030, -p68040, -pcpu32 (UNIX/DOS) option from the address of the word containing the displacement. For example, a call at address 0 (whose displacement is at address 2) can only reference an address up to 32769, but a call at address 5000 (whose displacement is at address 5002) can reference an address up to 37769.

  The subroutine call at location 40000 (see Figure 8-4) cannot be made position independent because it attempts to reference an address that is not in the range of 7234 to 72769. The range is calculated using the following equations:

  40002 - 32768 = 7234

  40002 + 32767 = 72769

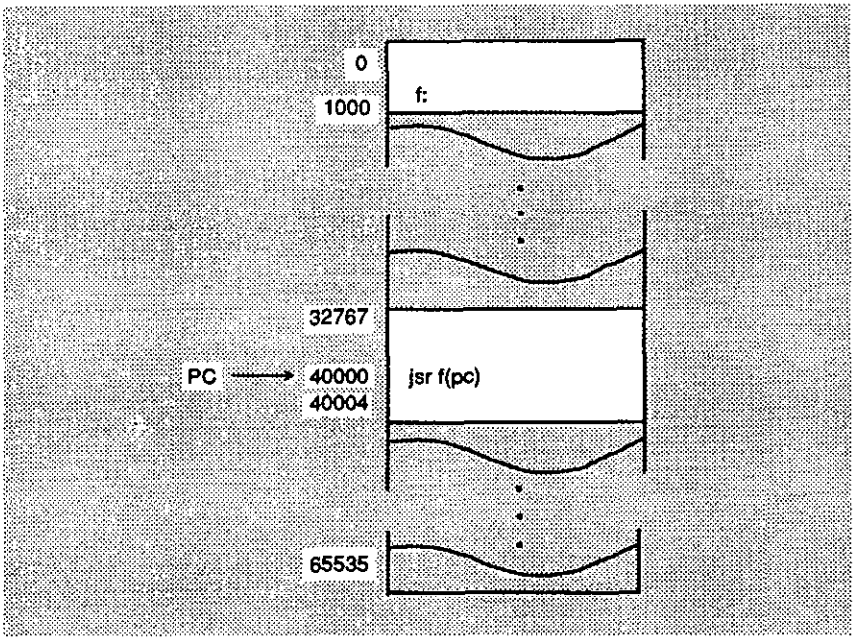

Figure 8-4. Position-Independent Code (Example 1)

To generate position-independent data, you must adhere to the following rules:

- Do not make data references to absolute data addresses within the program. For example, the following code cannot use register-relative data addressing because it uses an absolute data address:

```
/* reference to absolute data address */
read_from(0x02);/* read from data address 0x02 */
```

- Do not use pointers initialized to absolute data addresses within the program. For example, the following code fragment cannot use register-relative data addressing because it initializes pointers to absolute data addresses:

```
/* pointers initialized to absolute addresses
   of strings */
char *table[3] = {"table", "of", "strings"};
/* pointer initialized to absolute address
   of data */
char c;
char *p_c = &c;
```

- Do not locate a data reference and its target too far away from each other. The target of a data reference must be in the range of -32768 to 32767 bytes or -2147483648 to 2147483647 bytes if you use the /long (VMS) or -Ml (UNIX/DOS) option and /cpu=68020, /cpu=68030, /cpu =68040, /cpu=cpu32 (VMS), or -p68020, -p68030, -p68940, -pcpu32 (UNIX/DOS) option from the address of the word containing the displacement. For example, an instruction at address 0 (whose displacement is at address 2) can only reference an address up to 32769, but an instruction at address 5000 (whose displacement is at address 5002) can reference an address up to 37769.

  The instruction at location 40000 (see Figure 8-5) cannot be made position independent because it attempts to reference an address that is not in the range of 7234 to 72769. The range is calculated using the following equations:

  40002 - 32768 = 7234

  40002 + 32767 = 72769

  If you are using A*n*-relative data, the default linker command file must be modified.

- The default linker command file contains the following:

```
; if using A5-relative data addressing, enable the next 2 lines:
;
; index ?a5, vars ; A5 will be pointed to sect vars
; load mcc68ka5XX.lib
```

You need to make your own command file by:

1. Copying the default linker command file to your directory.
2. Uncommenting the `index` and `load` commands.
3. Specifying the correct library and path (see the *User's Guide* for the location of the libraries on your host).
4. Invoking mcc68k with the -e option (see *Pass Command File To Linker* in the *User's Guide*).

Note

Refer to the section titled *Code Organization* later in this chapter for information on the location and addressing of the various code and data sections.

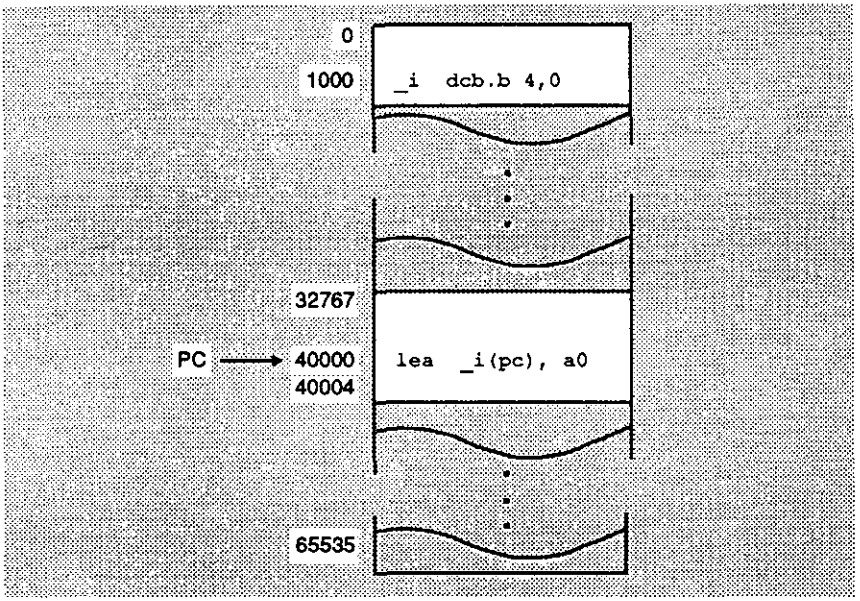

Figure 8-5. Position-Independent Code (Example 2)

Assembler and Linker Considerations

The assembler and linker automatically handle position-independent code; therefore, no special assembler or linker command line options need to be specified. When code is made position independent, the resident run-time linker adds the address assigned to the first instruction to all the load addresses in the executable allowing the program to execute at these new locations.

Absolute Versus Register-Relative Addressing

The following program fragment example is used to show the different assembly language output generated by the compiler depending on the /code=pc, /data=pc, /data=A*n* (VMS) and -Mcp, -Mdp, and -Md*n* (UNIX/DOS) options.

```
int data_1; /* data_1 is a global variable */
main()
{
      function_1();
}
function_1()
{
      data_1 = 3;
}
```

The code generated as shown in Table 8-7 is position dependent when absolute addressing is used; it is position independent when register-relative addressing is used.

Table 8-7. Absolute Versus Register-Relative Addressing

Position-Dependent Code (CODE=ABS and DATA=ABS)	Position-Independent Code (CODE=PC and DATA=PC)
... L1: jsr _function_1 ... _function_1: ... L2: moveq #3, d0 move.l d0, _data_1 ...	... L1: jsr _function_1(pc) ... _function_1: ... L2: lea _data_1(pc), a0 move.l #3, (a0) ...

Absolute Addressing

Figure 8-6 shows absolute addressing for code and data. The JSR (jump to subroutine) instruction uses an absolute code address, and the MOVE.L (move long) instruction uses an absolute data address. The absolute addresses immediately follow the operation codes in memory. Absolute addresses are also used as effective addresses.

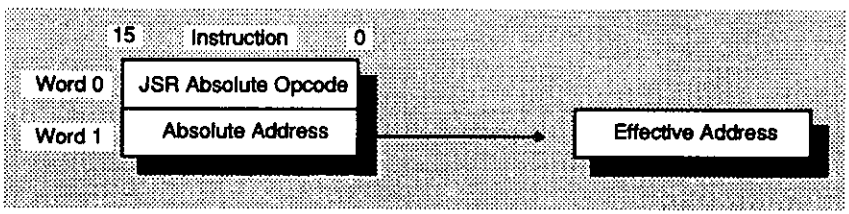

Figure 8-6. Absolute Addressing for Code and Data

Figure 8-7 shows how absolute addressing will position code in memory. Table 8-8 shows the commands for this example. The base address (BA) is the address where the program is loaded (i.e., fixed at link time). When the target of the JSR is computed, the program counter (PC) contains the address of the word containing the absolute address.

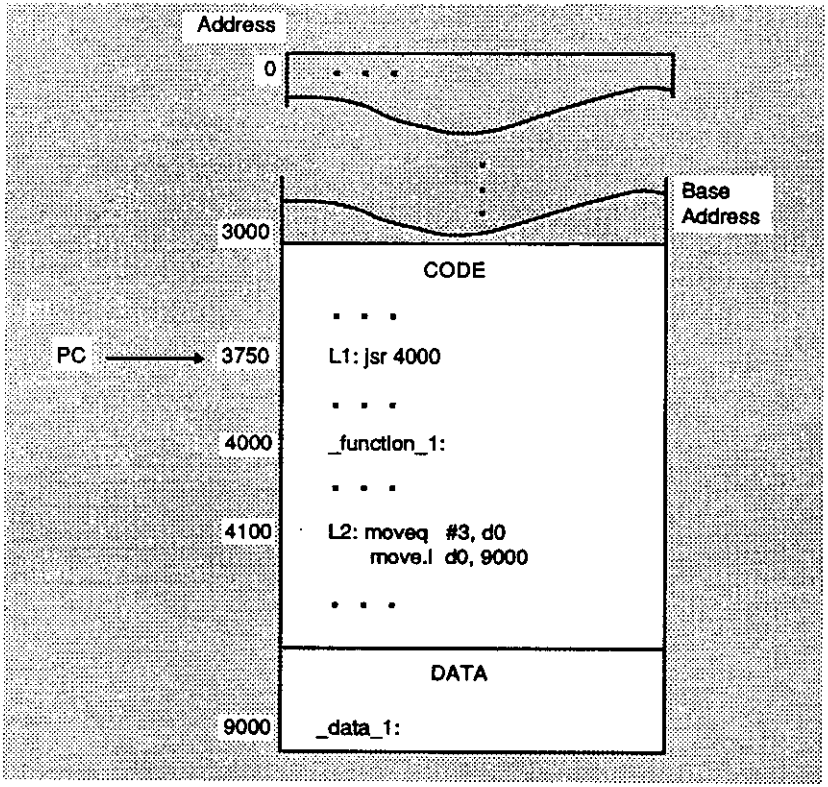

Figure 8-7. Absolute Addressing Example

Table 8-8. Commands for Absolute Addressing Example

PC	Description
3750 = L1	Calls `_function_1:` `jsr _function_1`
4000 = `_function_1`	The next section of code "..." is executed

4100 = L2	The value 3 is assigned to `_data_1:`
	`moveq #3,D0`
	`move.l d0,_data_1`

PC-Relative Addressing

Figure 8-8 shows PC-relative addressing for code and data. The JSR (jump to subroutine) instruction uses a PC-relative code address, and the LEA (load effective address) instruction uses a PC-relative data address. The effective address is determined by adding the contents of the program counter register to the displacement (or offset). When the target of the JSR is computed, the program counter contains the address of the word containing the offset.

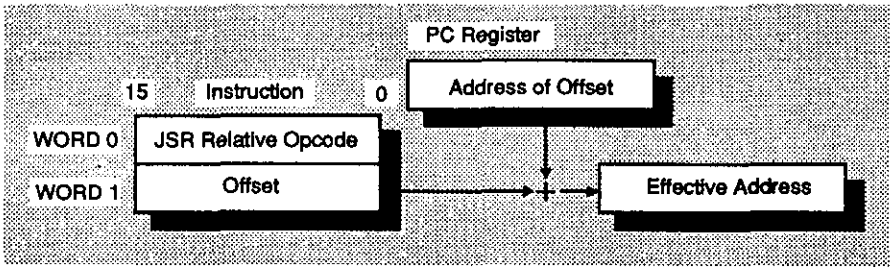

Figure 8-8. PC-Relative Addressing

Figure 8-9 illustrates how PC-relative addressing will position code in memory. Table 8-9 shows the commands for this example. The base address is the address where the program is loaded (i.e., determined at load time). Regardless of where the program is loaded, the relative positions of code and data remain the same.

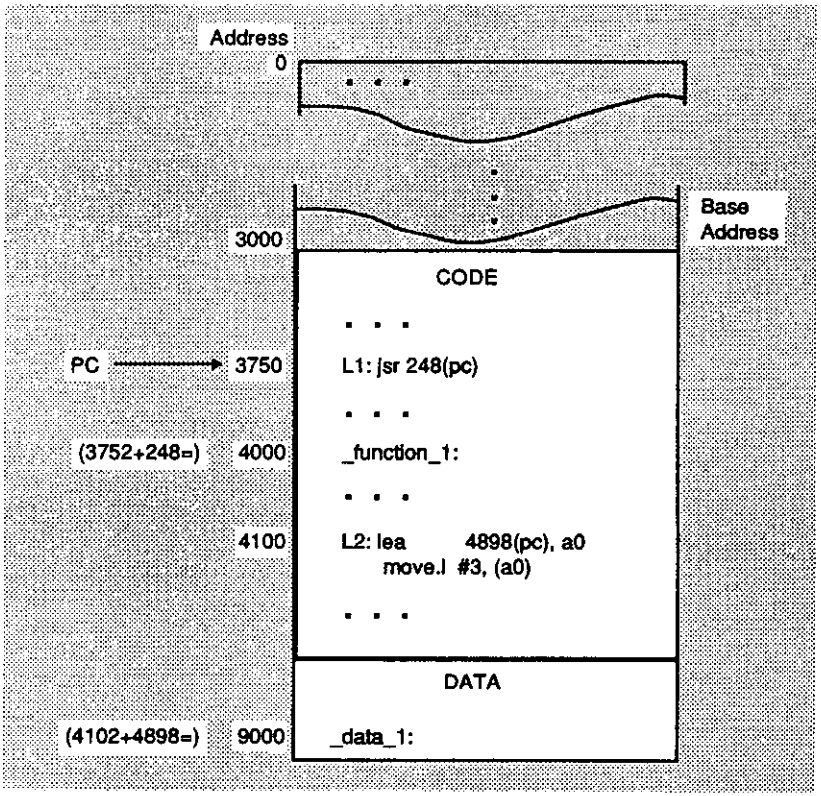

Figure 8-9. PC-Relative Addressing Example

Table 8-9. Commands for PC-Relative Addressing Example

PC	Description
3750 = L1	Calls _function_1: jsr _function_1(pc)
3752 + 248 = 4000 = _function_1	The next section of code ". . ." is executed
4100 = L2	The value 3 is assigned to _data_1: lea _data_1(PC),a0 move.l #3,(a0)

Reserving a Register

Since it can be useful to reserve a register for shared data, system data, reentrant programs, or shared programs, the Microtec Research C compiler allows you to reserve register **A2**, **A3**, **A4**, **A5**, or **A6**. Only one register can be reserved during a given compilation. The compiler does not generate any code that uses the reserved register. If the **A6** register is reserved, **A5** will be used as the frame pointer.

A reserved register is usually used in conjunction with a multitasking environment. A Multitasking Environment (MTE) handles one or more programs executing simultaneously. A MTE can be thought of as an operating system; however, it might also be part of the application program or part of an embedded system.

To reserve a register, use the /reserve (VMS) or -Mr*n* (UNIX/DOS) option described under *Generate Position-Independent Code and Data* in the *User's Guide*. Enter the option on the command line followed by the register to be reserved (**A2**, **A3**, **A4**, **A5**, or **A6**). Registers **A0**, **A1**, and **A7** can never be reserved since the compiler uses them for special purposes. When this option is used, the generated code can be slightly less efficient in some cases.

Shared Program

When a register is reserved for a shared program, a single physical copy of the program can be shared by one or more users simultaneously. These can be single-tasking, multitasking, or interrupt-based programs, but only local variables and ROM variables (global or static) can be used. In this case, the reserved register is used as a pointer to the currently used data area (it points to only one data area at a time). Shared programs must be reentrant. They are shown in Figure 8-10.

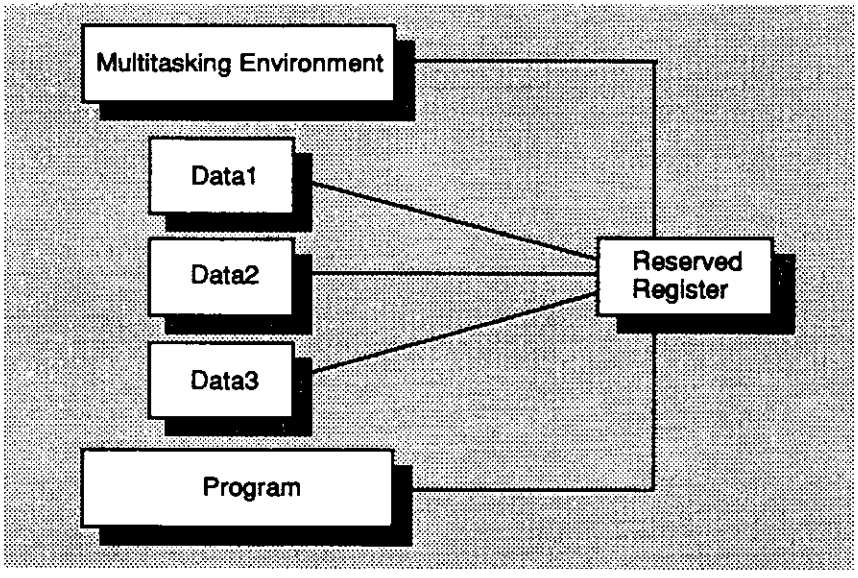

Figure 8-10. Shared Programs

Shared Data

When a register is reserved for shared data, a single physical copy of data in a multitasking environment can be shared among more than one program. The reserved register is used as a pointer to the shared data area as shown in Figure 8-11.

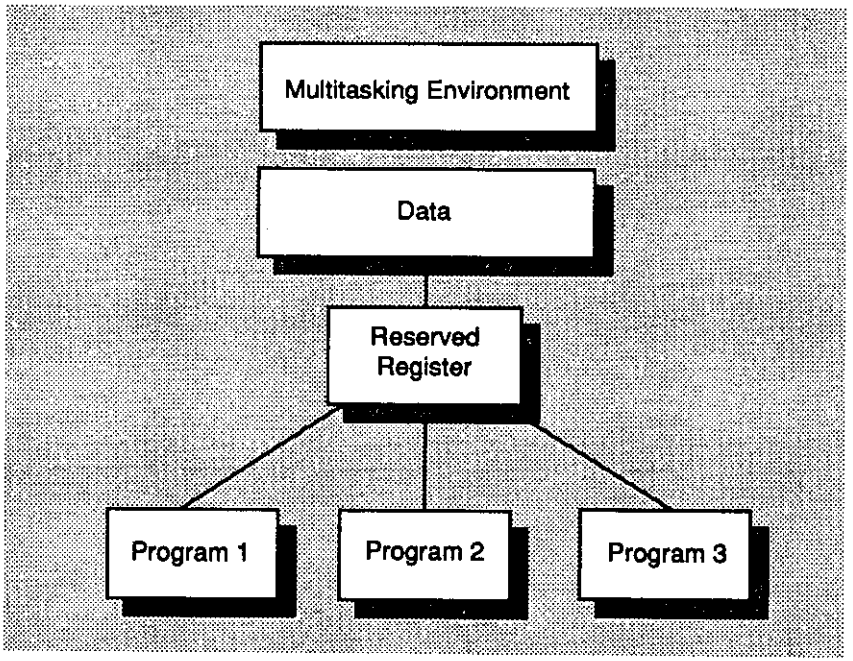

Figure 8-11. Shared Data

System Data

When a register is reserved to point to a system data area, special system data can be manipulated by a program. The Multitasking Environment (MTE) sets up the system data area before the program is loaded. The MTE must also load the address of the system data area into the reserved register. This is shown in Figure 8-12.

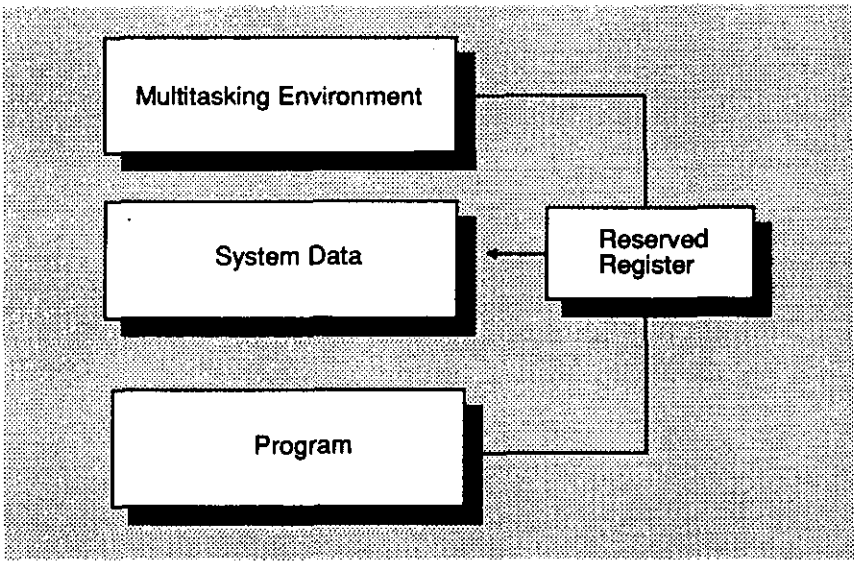

Figure 8-12. System Data

Special Purposes

It can be useful to reserve a register for special purposes. For example, a register can be used exclusively by an assembly language routine that is to be linked with C code. A register might also be reserved for a fast recall of a certain address or value, for locating data for ROM-based routines, and for memory management.

When a register is reserved for special purposes, it can either point to data or store the area that contains a special value. In Figure 8-13, a ROM-based routine reserves a register for a "hook" to find its data.

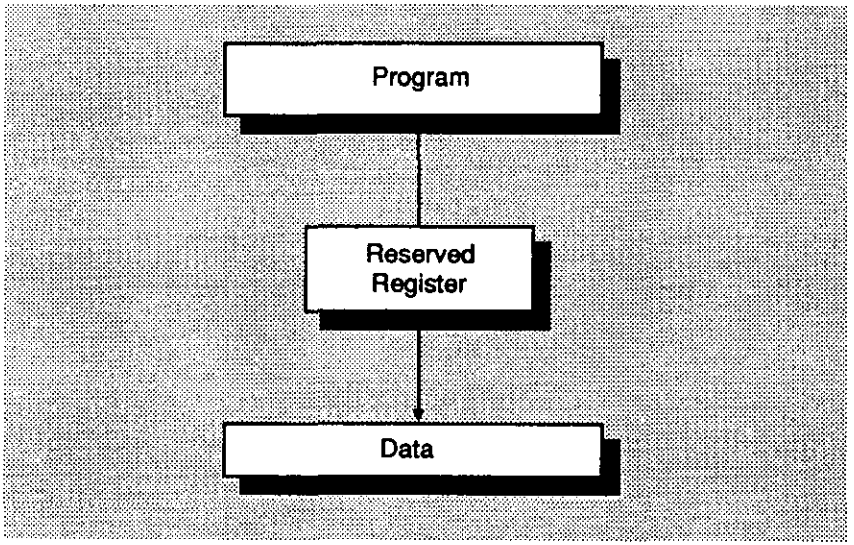

Figure 8-13. Reserved Register as a Pointer to Data

In Figure 8-14, the reserved register acts as a storage area containing a value used by the assembly language routine.

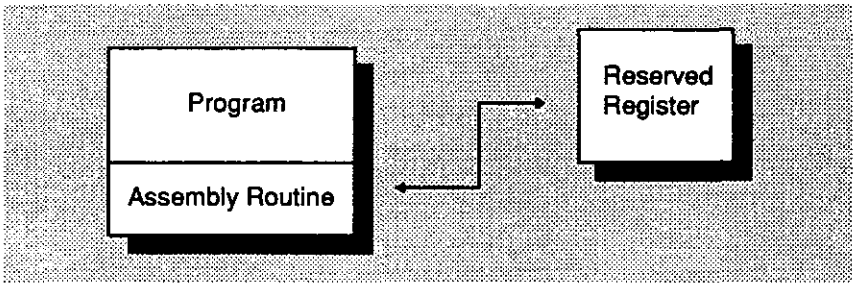

Figure 8-14. Reserved Register as a Storage Area

Reentrant Code

A routine is reentrant if it can be interrupted during its execution and re-invoked by subsequent calls (for example, from an interrupt service routine) any number of times. When each subsequent call has completed its invocation, the prior invocation resumes processing where it left off without loss of data.

The code generated by the compiler will be reentrant provided that the following rules are followed:

1. Do not write to global variables.
2. Do not write to local static variables.
3. Use the non-reentrant library routines listed in Chapter 5 only in the manner described below.

For the purposes of this discussion, a thread of computation, or simply "thread", is a given sequence of instructions. A thread can be interrupted by an external interrupt, and control given to another thread at almost any time. Multiple threads may access the same executable code at any given time. An interrupt service routine, while it is active, is also a thread of computation

Interrupt service routines, when they are interrupted and re-entered by another interrupt, constitute multiple threads of computation. In a multiprocessing or multitasking environment, each process or task is a thread of computation. A reentrant routine is one which will function correctly even though multiple threads may be executing it simultaneously.

In general, the I/O routines which are declared in stdio.h are non-reentrant. This is due to the fact that they modify elements of the array _iob which is declared in stdio.h. However, the three string I/O functions sprintf, vsprintf, and sscanf are reentrant. Therefore, the following code may be used for writing formatted output:

```
i = sprintf(buff, format_string, arguments);
write(1, buff, i);
```

Description:

`sprintf`	Returns the number of characters written.
buff	Is an area of memory where the formatted output will temporarily be written.
format_string	Is a printf format string.
arguments	Is the arguments used by *format_string*.
`write`	Outputs *i* bytes from *buff* to the output port.
`1`	Causes *write* to write to stdout.

There is another approach to using I/O routines when reentrancy is required. If each thread performs I/O through a unique FILE * or "stream", the I/O routines can be considered reentrant. This is due to the fact that each thread is modifying its own unique element of the _iob array. However, because fopen must scan the _iob array to find an unopened element, each stream must be opened in the application's start-up code before individual threads are activated.

Functions which modify errno are also non-reentrant. Many of the functions declared in math.h modify this variable, along with the functions strtod, strtol, strtoul, and others. If a multi-threaded environment is required, the easiest way of using these routines is simply not to use or modify errno in any user-written code.

The other functions listed as non-reentrant in Chapter 5 may be used in a multi-threaded environment; however, only one thread may call any group of functions accessing the same static variable. For example, only one thread may call the functions malloc, calloc, zalloc, realloc, and free. These routines may be called by multiple threads only by using one of the techniques listed below.

A Simple Solution

In situations in which multiple threads must call non-reentrant library functions, there is a solution which allows this to be done in simple situations. Basically, during a context switch, the variables in question are saved. When the thread is

resumed, the saved data is restored. This approach requires a memory storage area for each thread, although an interrupt service routine may be able to save and restore the data from the stack. Also, if a large amount of data must be stored, the save/restore operation may take a prohibitively large amount of time.

Multitasking/Multi-Threaded Environments

The above suggestions work around the reentrancy limitations of the libraries. If for one reason or another, these workarounds are not usable, then other solutions may be used. Two solutions are described here.

First Approach

In the first solution, user-written functions are compiled using absolute addressing. The vars and zerovars sections are renamed and the A5 register is reserved for compilations. This forces all user static data to be placed into absolute locations in memory. These modules are then linked to the appropriate A5-relative addressing library. Memory for library functions is allocated dynamically at the beginning of each thread, and the A5 register is set to address this space. In order to actually accomplish this, you must:

1. Compile all user-written routines that are to be shared between threads using A5-relative addressing. Use of static data should be avoided in these routines. If static data is used, it must be uninitialized. The vars section must be empty; the initdata command of the linker will not initialize this data correctly.

2. User modules that are not shared between threads must be compiled for absolute addressing; the A5 register is reserved and the vars and zerovars sections are renamed. Weak externals must not be used in these modules. This may be done by using the options:

   `-Mr5 -Xp -NZ`*`userzero`* `-NI`*`uservars`*

   Unless that variable is intended to be used in inter-thread communication, no task should use a static variable name that is the same as a static variable name used by another thread.

3. Modify the code in entry.c to dynamically allocate memory for each thread for the zerovars section created by the libraries and any user data created in Step 1. If a thread requires stack space, it must be allocated at this time also. Set non-stack memory to zero at start-up time.

   The example code in entry.c for support of reentrant libraries shows how memory might be allocated for one thread.

4. If static data from a library function or a user-written shared function must be accessed by a function compiled with absolute addressing, one of the following techniques may be used:

   - Compile a function which returns the address of the data in question. Compile the function as in Step 1 using A5-relative addressing. Call this function from the function compiled with absolute addressing to determine the address of the data.

   - Use a macro that inserts assembly code using A5-relative addressing. A macro to access **errno** might be written as follows:

```
#define errno (*(ASM (" xref 'errno'"),\
      ASM (" lea ('errno').w(a5),a0"),\
      ASM (int *," move.l a0,d0")))
```

5. Modify the routine **entry.c** to allocate memory for the libraries at program startup. The A5 register should be set to address this memory. Memory must be allocated for each thread. If a thread requires its own stack space, this space must be allocated at the same time. An example of how this might be done is shown in **entry.c**.

6. If the **initdata** feature of the linker is to be used, recompile the Microtec Research-supplied routine **csys.c** for A5-relative addressing. Set the command line option **-D_INITDATA** to cause **csys.c** to initialize user static data. You can only used the **initdata** feature to initialize the static data in functions that are compiled using absolute addressing.

7. When linking, do not use the default linker command file. The wrong library will be linked if you do. Use an **index** command to inform the linker about A5 addressing, and use the **load** command to load the correct A5-relative addressing library. Use the **-e** option to specify the linker command file. The following is a good starting point for writing a linker command file:

```
listabs  publics,internals
listmap  publics
format   ieee
extern   ENTRY
index    ?A5,zerovars,$8000 ; entry.c will point A5
                            ; to section zerovars
initdata uservars
sectsize heap=$8000
sectsize stack=$1000
sect     zerovars=0
sect     literals=$10000
order    zerovars
order    literals,strings,const,initfini,code  ; ROM
order    ??INITDATA                            ; ROM
order    userzero,uservars,ioports,heap,stack  ; RAM
load     /usr/mri/lib/mcc68ka5XXX.lib          ; library
```

8. At runtime, when a context switch occurs, the A5 register is set to address the data space of the thread which is about to run.

The memory configuration used by this technique is shown in Figure 8-15.

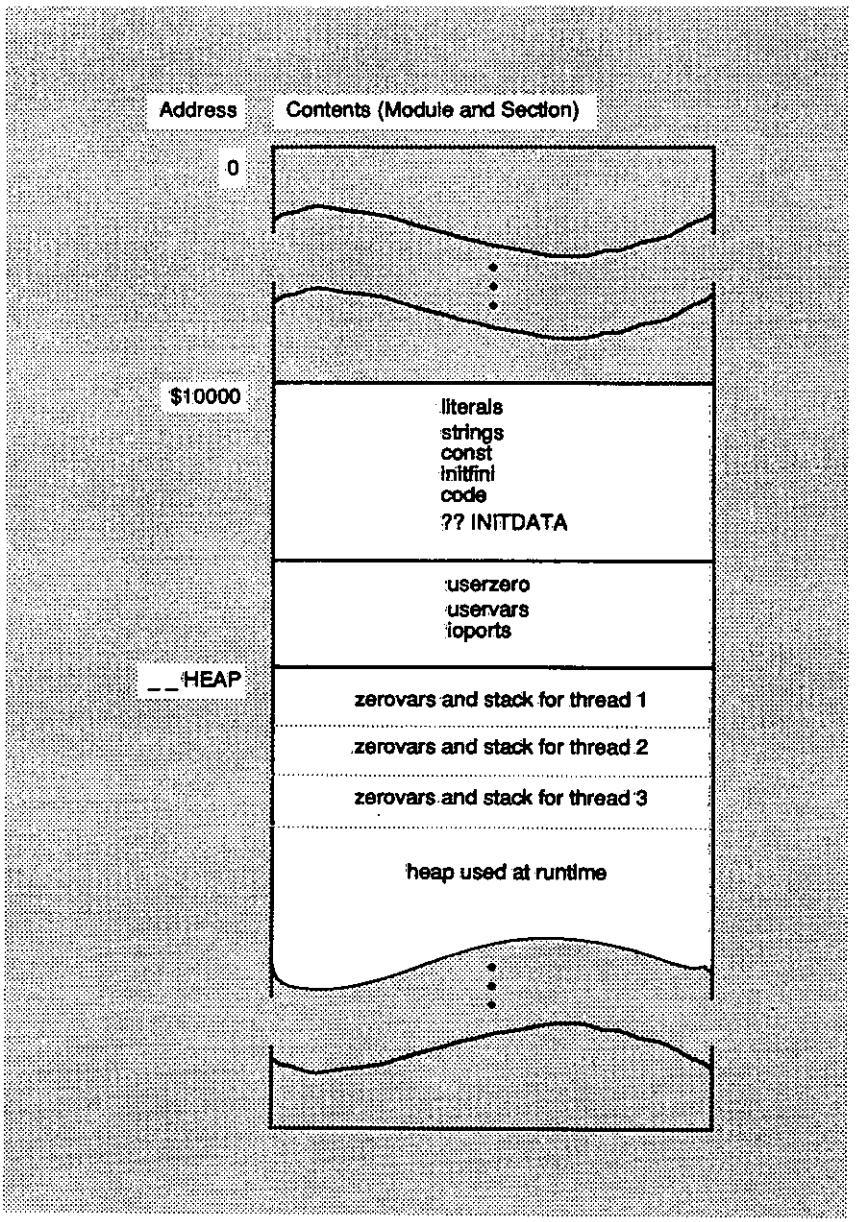

Figure 8-15. Memory Configuration for Multi-Threaded Environments (Approach 1)

Second Approach

In the second solution, the approach is to compile user-written functions using A5-relative addressing. Initialized static data cannot be used, so the vars section is empty. The zerovars section is renamed to a name which is unique to each thread. At link time, these sections are then positioned to begin at the same memory location. Then, at runtime, memory is allocated for the data in each thread in your start-up routines and the data is referenced by A5 addressing. In order to actually accomplish this, you must:

1. Compile all your modules using A5-relative addressing and rename the zerovars section to a name unique to each thread. Do not define any initialized data in these modules as the initdata feature of the linker does not work well with this technique. In other words, the vars section must be empty for all these modules. There should be no common static variable names between threads. Weak externals must not be used. Compiler options which might be used to do this are:

   ```
   -Md5 -Mcp -Nz task1zero -Xp
   ```

2. Modify the code in entry.c to dynamically allocate memory for each thread for both a zerovars section created by the libraries and the user data sections created in Step 1. If a thread requires stack space, it must be allocated at this time also. Set non-stack memory to zero at start-up time.

   The example code in entry.c for support of reentrant libraries must be modified for use with this approach.

3. Link to the appropriate A5-relative addressing library. Set the starting point address of each data section created in Step 1 to the same value. The linker will generate Section Overlap error messages caused by the placing of multiple sections in the same memory space, but these may be disregarded. The following can be used as a good starting point for writing a linker command file:

```
listabs  publics,internals
listmap  publics
format   ieee
extern   ENTRY
index    ?A5,zerovars,$8000    ; entry.c must set A5
                               ; to section zerovars.
sectsize heap=$8000
sectsize stack=$1000
sect     zerovars=0            ; position zerovars sect.
sect     task1zero=554         ; position data for each
sect     task2zero=554         ; thread at the same memory
sect     task3zero=554         ; location.
sect     literals=$10000
order    literals,strings,const,initfini,code  ; ROM.
order    ioports,heap,stack                    ; RAM.
load     /usr/mri/lib/mcc68ka5XXX.lib          ; library.
```

4. At runtime, when a context-switch occurs, the A5 register is set to address the data space of the thread which is about to run.

The memory configuration used by this technique is shown in Figure 8-16.

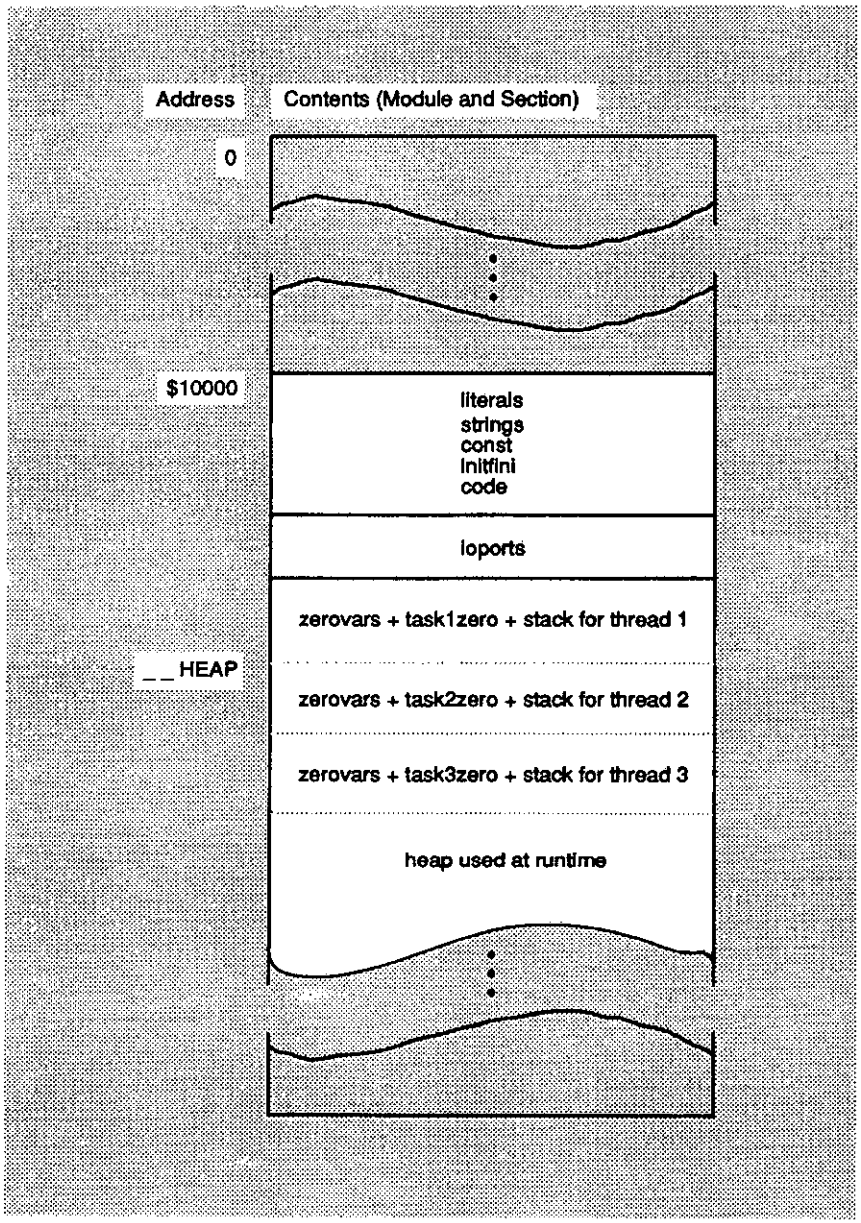

Figure 8-16. Memory Configuration for Multi-Threaded Environments (Approach 2)

Direct Memory and I/O Port Addressing

You can examine and modify absolute memory locations by defining a macro that functions exactly like a normal C variable, except that it is located at a particular memory address given by a number. For example, a macro called `memloc` is defined in the following example. This macro will represent the contents of the byte at location 1A (hexadecimal). The macro definition is:

```
#define memloc *(char *)0x1a
```

Whenever `memloc` appears in the program, it is replaced with the macro definition text ***(char *)0x1a**, which means "1a hexadecimal, treated as a pointer to a character and dereferenced". If `memloc` appears in an expression or on the right-hand side of an assignment, it represents the contents of byte location **0x1a**. If `memloc` appears on the left-hand side of an assignment, it represents the byte address **0x1a**.

The macro `memloc` can be used like an ordinary byte variable:

```
chr1 = memloc + 1;
memloc = 'a';
```

The macro `memloc` can also correspond to a memory-mapped I/O location, in which case each use would have the side effect of reading or writing.

Calling Functions at an Absolute Address

It is recommended that you use the asm pseudo-function to call a function at an absolute address. However, if you are concerned about porting your program to a different processor, use the technique described here.

This section describes how a C program can call an independently compiled and linked function which is placed at an absolute address without having to use assembly language. You can use this technique when an application program is required to call a system routine at a fixed ROM address.

To call a function at an absolute location, you can define a macro which declares a function.

Example:

```
#define ABS_FUNC (*(int (*)())0x124)
```

The macro defines `ABS_FUNC` to be an integer-valued function at location `0x124`. You can then call `ABS_FUNC` just like an ordinary function:

```
i = ABS_FUNC(); /* ABS_FUNC can be in any expression */
```

If parameters are being passed, the routine you are calling must have a compatible calling sequence. The return value for integer functions is placed in register D0. If a return value is expected, this is where the called function must place it. For more information on function parameter passing and return values, see Chapter 7.

Data Initialization

Variable values can be initialized at runtime or compile time. This example shows run-time and compile-time initializations:

Example:

Run-Time Initialization	Compile-Time Initialization
`int i;` `i = 15;`	`int i = 15;`

Code is always ROMable when variables are initialized at runtime. The constant value shown on the left is stored in the code section.

When a variable is initialized at compile time, the compiler allocates the variable, along with its value, in the vars section. Often the vars section will be located in RAM. For most ROM-based embedded systems, this presents a problem. When your program starts executing, RAM is typically not initialized with the variable values contained in the data section.

Saving Initialized Variables in ROM

Because RAM is typically not initialized with the variable values contained in the data section when your program starts executing, many embedded programmers use the convention that only constants can be initialized on the variable declaration. The value of the variables cannot be changed at runtime. In this case, the vars section can be placed in ROM, along with the code section.

If you do not want variables initialized at compile time on the data declaration to be constants, you can use the INITDATA facility in most ROM/RAM systems. This facility initializes the values of these variables when the program restarts.

Using the INITDATA Command (IEEE Only)

For information on INITDATA for systems which generate COFF, see the *ASM68K Reference Manual.*

The INITDATA facility initializes variables at program start-up time which have been initialized by you through variable declaration. The facility makes a copy of the variable's values and places them in the ??INITDATA section. During the run-

time initialization routine, the values are copied into RAM. The compiled program then accesses the variables in RAM and can change their values.

To use the INITDATA command:

1. Edit your linker command file and insert this command:

```
initdata vars
```

2. Edit the file csys.c and uncomment this line:

```
initcopy()
```

3. Reassemble the file and link it in before the library.

Code Organization

The compiler and linker allocate code and data to the sections listed in Table 8-10.

Table 8-10. Code and Data Allocation

Contents	Section Name	Addressed As:
code	code	code (ROM)
uninitialized data	zerovars	data (RAM)
initialized data	vars	data (RAM)
user const vars	const	code (ROM)
strings	strings	code (ROM)
compiler-generated literals	literals	code (ROM)
compiler-generated tags	tags	data (RAM)

If a variable is not initialized in any module, it is known as a "weak external". By default, weak externals are treated as "C common". The linker will allocate "C common" variables in the zerovars section. The compiler will allocate uninitialized static variables in the zerovars section. The /weakextern (VMS) and -X (UNIX/DOS) options can be used to modify this behavior. For more information, refer to the *Initialize Uninitialized Global Data* options in the *User's Guide*.

Changing Default Section Names

Refer to the options in *Name the Sections* in the *User's Guide* for information on specifying alternate section names.

Changing Default Addressing

The code section should always be located in ROM. Items in the code section are referenced according to the address mode chosen with the code option (absolute or PC-relative) described in *Generate Position-Independent Code and Data* in the *User's Guide*.

The zerovars section should always be located in RAM. Items in the zerovars section are referenced according to the address mode chosen with the data option (absolute, PC-relative, or A*n*-relative) described in *Generate Position-Independent Code and Data* in the *User's Guide*.

Refer to the options in *Generate Prologue/Epilogue Protocol and Code Variations* in the *User's Guide* for information on the tags section. The tags section should always be located in RAM. Items in the tags section are referenced according to the address mode chosen with the data option (absolute, PC-relative, or A*n*-relative) described in *Generate Position-Independent Code and Data* in the *User's Guide*.

The remaining sections (vars, const, strings, and literals) can each be located in ROM or RAM at link time. If you are generating position-independent code or data and you will be locating any of these sections in the non-default area (ROM/RAM), you must tell the compiler how to address these sections.

If a section is to be placed in ROM (with the code section), its items should be referenced according to the code option described in *Generate Position-Independent Code and Data* in the *User's Guide*. If a section is to be placed in RAM (with the zerovars section), its items should be referenced according to the data option described in *Generate Position-Independent Code and Data* in the *User's Guide*. Refer to *Choose Address Mode for Sections* in the *User's Guide* for information on specifying alternate addressing modes for the various sections.

Optimizations 9

General Optimizations

The MCC68K C Compiler performs several optimizations before generating code for a C program. These optimizations usually involve constant expressions, or expressions with known results which can be evaluated at compile time. Normally, programmers seldom write explicit expressions with only constants, but it is not uncommon for them to write an expression with defined constants.

Algebraic Simplification

These optimizations replace an expression with an equivalent but simplified expression.

Examples:

```
a * 1   ⟶  a
b - 0   ⟶  b
c == c  ⟶  1
x -= x  ⟶  x = 0
```

Constant Folding

Arithmetic and logical operations involving only constants are evaluated at compile time.

Example:

```
i = 1 + 2 * 3 + 4          ⟶  i = 11
(a  >>  3)  >>  1          ⟶  a >> 4
i + 5 + j - 6 + k - 7      ⟶  i + j + k - 8
(4 & 1) || (5 > 7)         ⟶  0
```

Strength Reduction

These optimizations replace operations with equivalent but less expensive (reduced strength) operations.

Examples:

```
a * 128              ⟶  a << 7
(unsigned) b / 16    ⟶  b >> 4
(unsigned) c % 32    ⟶  c & 31
```

Redundant Code Elimination

Code sequences that produce no real side effects are considered redundant and can be eliminated. They usually evolve as the result of other optimizations.

Examples:

```
a = a + 0;    ⟶  No code generated
b + c;        ⟶  No code generated
x &= x;       ⟶  No code generated
```

Unreachable Code

Unreachable code sequences are usually detected after performing jump optimizations. Unlabeled instructions immediately following an unconditional jump may be removed. Conditional expressions which can be reduced to constants also produce unreachable code.

Examples:

```
    goto l1;
    i1 = 3;  ┐
    i2 = 4;  ├──⟶ No code generated
    i3 = 4;  ┘
l1:
    i4 = 3;

    a = ( 3 > 0 ) ? xx (6) : yy (7);  ⟶  a=xx(6);
    if ( a != a ) b = 5;              ⟶  No code generated
```

Global Optimizations

The Microtec Research C compiler performs several optimizations that make programs smaller and faster by applying global data flow and control flow analysis on the entire function.

Register Coloring

Register coloring optimizes register usage throughout an entire routine. It is particularly valuable for keeping most of the local variables in the registers at all times. Using data flow analysis, the compiler finds the lifetime of each variable. The register coloring algorithm can then increase the number of variables which are stored in registers by using the same register for several variables in the same routine.

Factorization

Factorization optimization is particularly useful for storing a frequently used static address in a register. It references the static value or address through register indirect addressing and stores it at the beginning of the function. Factorization optimization is applied to the entire function.

Example:

```
test(1);      movea.l   #_test,a2
test(2);      pea       1
test(3);      jsr       (a2)
              pea       2
              jsr       (a2)
              pea       3
              jsr       (a2)
```

Dead Code Elimination

This optimization eliminates unreachable code (i.e., "dead code"). Unreachable code occurs through user error or "conditional compilation" techniques which exploit the preprocessor.

Example:

```
#define TRUE 1
#define FALSE 0
#define DEBUG 0

while (a)
{
      test1();
      if (b)
            continue
      else
            break;
      test2();                    ---> Dead code
}
if(DEBUG)                         ---> Dead code
      printf("debug1 \n");        ---> Dead code

while (TRUE)
{
      test1();
      test2();
}                                 ---> Dead code
```

Unused Definition Elimination

An unused definition is an assignment in which the left hand side variable is either never used, or used after being reassigned. The compiler eliminates unused definitions.

Example:

```
...
...
i = j + 1;  /* if the value of the variable i is never used
...            again, this statement will be eliminated */
...
```

Example:

```
i = j + 1;  /* this assignment will be eliminated since i
               will not be used before it is reassigned */
i = k + 1;
```

Global Constant Propagation

When a constant is assigned to a variable, the constant may be carried forward to the subsequent uses of the value of that variable until a new value is assigned to that variable. This optimization is performed across basic blocks.

Example:

```
;  var1 = 2;
       moveq       #2,d0
       move.l      d0,_var1      /* var1 is global variable. */
;  if (var1)                     /* Will not do test, var1 */
                                 /* is known to be 2. */
;      i = var1;                 /* 2 will be used instead */
                                 /* of var1. */
       moveq       #2,d1
;  f(var1,i);                    /* 2 will be used instead */
                                 /* of var1. */
       move.l      d1,-(sp)
       pea         2
       jsr         _f
       addq.l      #8,sp
```

Global Copy Propagation

When a variable that is "cheap" to access (e.g., a register variable) is assigned to a variable that is "expensive" to access (e.g., a global variable), the "cheap" variable may be carried forward to the subsequent uses of the "expensive" variable until a new value is assigned to the "expensive" variable. This optimization is performed across basic blocks.

Example:

```
;  var1 = i;                     /* i is in d2. */
       move.l      d2,_var1      /* var1 is global variable. */
;  if (var1)                     /* Will test i instead */
                                 /* of var1. */
       tst.l       d2
       beq.s       L2
;      f(var1+2);                /* Function will pass i+2 */
                                 /* instead of var1+2. */
       move.l      d2,d0
       addq.l      #2,d0
       move.l      d0,-(sp)
       jsr         _f
       addq.l      #4,sp
  L2:
```

Branch Tail Merging

The compiler combines common instructions preceding two jump instructions which jump to the same location.

In the example below, the three instructions following the jump instruction on the left were combined under the `L12` label.

Example:

```
jsr       _func1              jsr       _func1
addq.l    #1,d0         L12:  addq.l    #1,d0
add.l     d0,8(a6)            add.l     d0,8(a6)
bra.s     L1                  bra.s     L1
...                           ...
jsr       _func2              jsr       _func2
addq.l    #1,d0               bra.s     L12
add.l     d0,8(a6)
bra.s     L1
...
```

In the example below, the compiler will not generate code for two `i1=7` statements. Instead, it will generate code for the `i1=7` statement on line `9` and only a jump statement for the `i1=7` statement on line `4`.

Example:

```
1       if(i)
2       {
3           i2 = 3;
4           i1 = 7;
5       }
6       else
7       {
8           i2 = 21;
9           i1 = 7;
10      }
```

Local Optimizations

Local optimizations are generally applied to small sections of the generated code. Some of the jump optimizations described in this section can also be considered local optimizations.

Common Subexpression Elimination

Common subexpression optimizations remove expressions which have been previously evaluated.

Example:

```
if (a+b > 3)          move.l   d2,d0
      test(a+b);      add.l    d2,d0
                      move.l   d0,d2
                      moveq    #3,d1
                      cmp.l    d0,d1
                      bge.s    L1
                      move.l   d2,-(sp)
                      jsr      _test
```

Redundant Load and Store Elimination

Loading the same variable after storing it is redundant and can be eliminated. This optimization can also be applied to pushing the same variable onto the stack.

Example:

```
i = expression;       (code for expression)
j = i & 5;            move.l   d0,_i
                      move.l   _i,d0        Instruction eliminated
                      andi.l   #5,d0
                      move.l   d0,_j
i = expression;       (code for expression)
printf ("%d", i);     move.l   d0,_i
                      move.l _i,-(sp)  ⟶  move.l d0,-(sp)
```

Generating Code for the Switch Statement

The Microtec Research C compiler applies a special optimization to generate efficient code for the switch statement. Depending on the number of cases and the range of case values, the compiler uses one of the following methods to generate a code sequence for the switch statement:

- If there are three cases or less, a series of compare and jump instructions are generated. The order of comparing the cases is the same as they appear in the program. Therefore, you can improve the execution speed by placing the most frequently occurring case first.
- If the table is fairly dense (i.e., not too many holes), the compiler uses a jump table indexed by the switch value. The compiler generates a short sequence of instructions to scale the switch value; this value is compared to the size of the jump table before indexing. This method has the fastest execution speed of the three methods.
- If the jump table in the previous method is sparse, two parallel tables are constructed instead: one for the case values and one for the jump labels. The compiler generates a short sequence of instructions to linearly search the case value table for the switch value. If it finds the switch value, the table entry offset is used to index into the jump label table. The order of case values is either sorted or the same as they appear in the program, depending on their range.

The compiler estimates the code and table size required for each method described above, and then selects the one with the minimum code size. When optimizing for time, the compiler will favor selecting the second method (i.e., dense table).

Local Constant Propagation

When a constant is assigned to a variable, the constant may be carried forward to the subsequent uses of the value of that variable until a new value is assigned to that variable.

Example:

```
; var1 = 2;
      moveq       #2,d0
      move.l      d0,_var1      /* var1 is global variable. */
; i = var1;                    /* 2 will be used instead */
                                /* of var1. */
      moveq       #2,d1
; f(var1,i);                   /* 2 will be used instead */
                                /* of var1. */
      move.l      d1,-(sp)
      pea         2
      jsr         _f
      addq.l      #8,sp
```

Local Copy Propagation

When a variable that is "cheap" to access (e.g., a register variable), is assigned to a variable that is "expensive" to access (e.g., a global variable), the "cheap" variable may be carried forward to the subsequent uses of the "expensive" variable until a new value is assigned to the "expensive" variable.

Example:

```
;  var1 = i;                    /* i is in d1. */
      move.l      d1,_var1      /* var1 is global variable. */
;  f(var1+2);                   /* Function will pass i+2 */
                                /* instead of var1+2. */
   move.l         d1,d0
   addq.l         #2,d0
   move.l         d0,-(sp)
   jsr            _f
   addq.l         #4,sp
```

Loop Optimizations

Loop invariant analysis speeds up loops by eliminating computations of invariant expressions and addresses. These computations are moved out of the loop and the value is stored in a register.

This optimization is particularly valuable for removing array subscripting from a loop when the subscript is a variable or expression that is not modified in the loop.

Loop Invariant Code Optimization

The loop invariant code optimization searches loops for expressions which yield the same result regardless of the number of times the loop is executed. The optimization then places the expression before the loop.

Example:

```
while (i<10)          move.l   d3,d2
{                     add.l    d3,d2
   test(j+k+l);       add.l    d3,d2
   i++;               bra.s    L2
}               L1:
                      move.l   d2,-(sp)
                      jsr      _test
                      addq.l   #4,sp
                      addq.l   #1,d3
                L2:
                      moveq    #10,d0
                      cmp.l    d3,d0
                      bgt.s    L1
```

Strength Reduction and Index Simplification

Strength reduction and index simplification is also referred to as induction variable elimination. The values of induction variables increment or decrement by a constant value for each iteration of the loop. These variables are often used to count or index an array. Also, multiples of these variables can be computed. It is often possible to replace all but one through the process of induction variable elimination.

Example:

```
for (i=1; i<10; i++)          moveq    #1,d1
   ary[i] = i;                lea.l    -76(a6),a0
                         L1:
                              move.l   d1,(a0)+
                              addq.l   #1,d1
                              moveq    #10,d0
                              cmp.l    d1,d0
                              bgt.s    L1
```

Loop Rotation

In both the for and while loops, the controlling expression is located syntactically at the top of each loop. When translating these loops, the controlling expression is rotated to the bottom. This rotation reduces the number of instructions within the body of the loop.

The while loop on the left is translated as if it were written as the sequence of statements on the right.

```
while (expression)          goto test;†
{                           loop:
statements;                 {
}                           statements;
                            }
                            test: if (expression) goto loop;
```

The for loop on the left is translated as if it were written as the sequence of statements on the right.

```
for (expr1; expr2; expr3)   expr1;
{                           goto test; ††
statements;                 loop:
{                           {
                            statements;
                            }
                            expr3;
                            test: if (expr2) goto loop;
```

† If the compiler determines that *expression* is initially true, then this `goto` statement is suppressed.

†† If the compiler determines that *expr2* is initially true, then this `goto` statement is suppressed.

Array Operator Synthesis

Because the C language does not have an "array assignment" operator, for loops are often used to assign a value to all elements of an array, or to assign one array to another.

Example:

```
char arr[HI_BND], arr2[HI_BND];
for (i=0; i<HI_BND; i++)
      arr1[i] = 'a';
for (i=0; i<HI_BND; i++)
      arr2[i] = arr1[i];
```

Constructs like the above can result in very inefficient code because of the number of loop iterations required.

The Microtec Research C compiler recognizes and optimizes constructs of the following form:

```
for (i=const1; i<const2; i++)
      arr[i]=const3;
for (i=const1; i<const2; i++)
      arr[i]=arr2[i];
```

Depending on the number of elements to be assigned, the compiler may "unroll" the for loop, or may minimize the number of iterations required. For example, to initialize a character array, the compiler will use move.l rather than move.b, thereby requiring only twenty-five percent as many iterations.

Because of this type of optimization, debug-time stepping through this type of for loop will not work as expected. The for loop is considered to be a single statement. One XRAY Debugger step command will be sufficient to step over the entire for loop.

This optimization is tied to the /optimize=globalflow (VMS) and -Og (UNIX/DOS) compiler options.

Jump Optimizations

The Microtec Research C compiler performs jump optimizations after generating code for each function. The compiler examines every jump instruction and its target location to recognize specific patterns for optimization. Since most jump optimizations create opportunities for further improvements, they are applied iteratively until no further changes result.

Redundant Jump Elimination

A jump instruction that jumps to a location which immediately follows it can be eliminated. Redundant jumps usually appear after applying other jump optimizations.

Example:

```
        bra.s L3    /* this instruction will be eliminated */
L3:
```

Multiple Jump Optimization

Generated code for complex control flow statements contain jumps to jumps, jumps to conditional jumps, or conditional jumps to jumps. These multiple jumps can be re-routed to jump directly to the target locations.

Example:

```
        bra.s L1            bra.s L2
        ...                 ...
        bne.s L1            bne.s L2
        ...                 ...
L1:     bra.s L2        L1: bra.s L2
        ...                 ...
```

Code Hoisting

The Microtec Research C compiler performs a simple form of code hoisting to move instructions which are common at the beginning of two branches of a conditional jump to the location before the jump.

Example:

```
        cmp.l 12(ab),d0             cmp.l 12(ab),d0
        bge.s L3                    lea.l (a1,d1.l),a0

        lea.l (a1,d1.l),a0          bge.s L3
...                             ...
L3:     lea.l (a1,d1.l),a0      L3:
...                             ...
```

Cross-Jump Optimization

The compiler combines instructions when they appear both preceding a jump and at the beginning of its target location.

Example:

```
        move.l  d0,-(sp)
        jsr     _func
        bra.s   L1                      bra.s   L12
        ...                             ...
        move.l  d0,-(sp)        L12:    move.l  d0,-(sp)
        jsr     _func                   jsr     _func
L1:     ...                     L1:     ...
```

Short/Long Displacement Optimization

During code generation, the compiler does not know the exact displacement between a jump instruction and its target location. Many of these jump displacements are changed as a result of applying jump optimizations. As a last step in jump optimizations, the compiler determines the exact displacements and replaces them with the shortest instructions possible for conditional and unconditional jumps.

Example:

```
        cmpi.l  #20,d0                  cmpi.l  #20,d0
        bne.l   L1                      beq.l   L2
        bra.l   L2              L1:     ...
L1:     ...                             bra.s   L3
        bra.s   L3
        ...
```

Function In-lining

In-line expansion of a user function saves the overhead usually associated with function calls, parameter passing, register saving, stack adjustment, and value return. Sometimes, further optimizations become possible because the actual parameters used in a call become visible, and the side effects of function calls also become exposed to the caller.

Example:

```
func(char op, int a, int b)
{
      if (op == 1)
           return(a+b);
      else
           return(a-b);
}

main()
{
int x, y, z;

...
           ; Parameter passing of 1, x, y removed.
           ; Register saving in func removed.
           ; if (op == 1), comparison removed.
           ; Register restoring in func removed.
           ; Return value removed.
           ; Stack adjustment after call removed.
           ; The code generated is:

      z = func(1, x, y);            move.l   d3,d4
                                    add.l    d2,d4

}
```

Instruction Scheduling Optimization

Improved performance can be achieved by rearranging the instruction sequence to avoid stalling the instruction unit pipeline. This includes overlapping CPU instructions with FPU instructions, and inserting useful instructions between those instructions that could cause pipeline stall because of data dependence.

Machine-Dependent Optimizations

This section describes machine-dependent optimizations.

Generating Code for Function Prologue and Epilogue

The standard code sequences for function prologues and epilogues are described in Chapter 7. The Microtec Research C compiler generates the full prologue or epilogue only when it is absolutely necessary or when the /frames (VMS) or -Kf (UNIX/DOS) option is used.

A prologue will not be generated if a stack frame is not needed, if there are no local variables, or if the function does not return a structure.

If only one register needs to be saved, the compiler will use the move.l instruction instead of the movem.l instruction. This saves 6 cycles.

Grouping Stack Adjust Instructions

The calling convention requires the calling routine (i.e., caller) to remove the function arguments from the stack after the call. The stack pointer is adjusted by adding a constant which equals the total size of the function arguments pushed. The compiler will try to combine two or more stack adjust instructions by accumulating the adjustment to the last instruction.

Example:

```
...                         ...
jsr       _func1            jsr       _func1
addq.l    #4,sp             ...
...                         jsr       _func2
jsr       _func2            addq.l    #8,sp
addq.l    #4,sp
```

This optimization cannot be applied if the first call is one of the function arguments of the second call, or if any jump or label occurs between the two function calls. The compiler accumulates a maximum of 50 bytes to reduce the stack growth. This maximum is particularly important for recursive function calls.

You can disable this optimization by using the /optimize=nocombine_pops (VMS) or -nOc (UNIX/DOS) option.

Indexing Arrays

When indexing into arrays that are less than or equal to 64K bytes, the index expression will be computed in short rather than int. This will result in faster, but not smaller, code.

This optimization will not be applied to arrays of unknown size (which can be declared as [ ]), arrays of size 1 (sometimes used to denote an unknown size), or arrays of greater than 64K bytes.

Compiler Output Files 10

Assembler Source File

The Microtec Research C compiler produces an assembler source file that is in Microtec Research and Motorola-compatible format. Microtec Research also has several assembler directives which can optionally pass variable name and data type information from the compiler directly to the Microtec Research XRAY Debugger.

Advantages of Producing an Assembly File

You can examine and optimize the generated assembly code file by hand. Be careful when you modify the assembly code; an incorrect change can produce unpredictable results.

You can use an optional command line option to intermix the original C source statements as comments with the generated assembly code in the output file.

C variable and line numbers are mapped into symbol names and placed in the output object module as described below.

Variable Names

The Microtec Research C compiler alters the names of external, public, and static variables to avoid possible conflicts. These naming changes are described below.

External and Public Variable Names

The Microtec Research C compiler alters the names of all variables and functions declared in the C input source. Variables and functions are prefixed with a leading underscore to avoid possible conflicts with assembler reserved words. The options described under *Modify Naming Conventions* in the *User's Guide* can change this behavior. These options are not recommended because they prevent access to the standard C library functions (all library functions have a leading underscore).

For a C program to access an assembly language routine or data item, the item name must begin with an underscore and be declared as public in the assembly routine. The C program should refer to the assembly language item without the underscore.

The following example shows function naming and global data in C and assembly language.

Example:

C Program

```
int errflag;
extern task();

driver()
{
...
  task (i);

...
}
```

Assembly Language

```
        ...

        XCOM    _errflag,4
        SECTION code,,C
        XDEF    _task
        ...
_task:  ...
        move.l  _errflag,d0
        ...
```

The C compiler reserves some identifiers beginning with two underscores for internal use. Avoid using identifiers with two leading underscores in assembly routines. Also, avoid using identifiers with one leading underscore in C source files; these identifiers may conflict with internal names.

Some assemblers and linkers translate all lower-case letters to uppercase, or vice versa. Since the C language is case-sensitive, check your assembler and linker documentation for information on this topic.

The Microtec Research C compiler and assembler toolkits retain the original upper- and lower-case status of all letters within a symbol.

Static Variable Names

The compiler automatically prefixes a string to all static variable names declared within a function. The string is _.S*n*_, where *n* is a number. For example, if the variable name *var* has been declared within a function, it will be changed to _.S*n*_*var* in the generated assembly language code. If a static variable is declared outside of a function, the actual variable name is emitted in the assembly language as a label. This avoids possible name conflicts between these two types of static variables. In either case, variables declared as static are not globally visible and cannot be accessed by an external module.

Line Numbers

If you specify the command line option for debugging, the compiler generates actual C source lines and line numbers, and includes them as comments in the assembly language source file.

If the /debug=lines (VMS) or -Gl (UNIX/DOS) option is specified, the compiler generates line number symbols as labels for debugging purposes. These line number symbols have the form LL*nn*, where *nn* is the line number. Line number information is generated for all executable lines of code.

Code and Data Sections

See the section *Code Organization* in Chapter 8, *Embedded Environments*, for information on the section names, placement, and method of addressing.

Compiler Output Listing

The compiler can optionally generate an output listing file using a command line option. When a listing is generated, the compiler assigns a line number, beginning with number 1, to each source line in your C module. When files are included in the C source module, the compiler will also assign a number, beginning with number 1, to the include file lines. When the last line of the include file has been numbered, the compiler resumes numbering the source file lines from where it left off.

The compiler output lists error messages if any occurred during compilation. An arrow points to the error in the listing. The error message is printed immediately below the line where the error was encountered.

The output listing also includes a code summary for each routine and the entire module. These code sizes are estimates since it is not known at compile time whether certain address references will require 16 or 32 bits.

ASCII Character Set: Appendix A

Table A-1. ASCII Character Set

Graphic	Octal	Numeric Decimal	Hex	Description
	00	0	0	NULL
	01	1	1	Start of heading
	02	2	2	Start of text
	03	3	3	End of text
	04	4	4	End of transmission
	05	5	5	Inquiry
	06	6	6	Acknowledge
	07	7	7	Bell (alert)
	010	8	8	Backspace
	011	9	9	Horizontal tab
	012	10	A	Line feed
	013	11	B	Vertical tab
	014	12	C	Form feed
	015	13	D	Carriage return
	016	14	E	Shift out
	017	15	F	Shift in
	020	16	10	Data link escape
	021	17	11	Device control 1
	022	18	12	Device control 2
	023	19	13	Device control 3
	024	20	14	Device control 4
	025	21	15	Negative acknowledge
	026	22	16	Synchronous idle
	027	23	17	End of block

(cont.)

Table A-1. ASCII Character Set (cont.)

Graphic	Octal	Numeric Decimal	Hex	Description
	030	24	18	Cancel
	031	25	19	End of medium
	032	26	1A	Substitute
	033	27	1B	Escape
	034	28	1C	File separator
	035	29	1D	Group separator
	036	30	1E	Record separator
	037	31	1F	Unit separator
	040	32	20	Space
!	041	33	21	Exclamation point
"	042	34	22	Quotation mark
#	043	35	23	Number sign
$	044	36	24	Dollar sign
%	045	37	25	Percent sign
&	046	38	26	Ampersand
'	047	39	27	Apostrophe or single quote
(	050	40	28	Opening (left) parenthesis
)	051	41	29	Closing (right) parenthesis
*	052	42	2A	Asterisk
+	053	43	2B	Plus
,	054	44	2C	Comma
-	055	45	2D	Hyphen (minus)
.	056	46	2E	Period (decimal)
/	057	47	2F	Slash
0	060	48	30	Zero
1	061	49	31	One
2	062	50	32	Two
3	063	51	33	Three

(cont.)

Table A-1. ASCII Character Set (cont.)

Graphic	Octal	Numeric Decimal	Hex	Description
4	064	52	34	Four
5	065	53	35	Five
6	066	54	36	Six
7	067	55	37	Seven
8	070	56	38	Eight
9	071	57	39	Nine
:	072	58	3A	Colon
;	073	59	3B	Semicolon
<	074	60	3C	Less than
=	075	61	3D	Equals
>	076	62	3E	Greater than
?	077	63	3F	Question mark
@	0100	64	40	Commercial at
A	0101	65	41	Upper-case A
B	0102	66	42	Upper-case B
C	0103	67	43	Upper-case C
D	0104	68	44	Upper-case D
E	0105	69	45	Upper-case E
F	0106	70	46	Upper-case F
G	0107	71	47	Upper-case G
H	0110	72	48	Upper-case H
I	0111	73	49	Upper-case I
J	0112	74	4A	Upper-case J
K	0113	75	4B	Upper-case K
L	0114	76	4C	Upper-case L
M	0115	77	4D	Upper-case M
N	0116	78	4E	Upper-case N
O	0117	79	4F	Upper-case O

(cont.)

Table A-1. ASCII Character Set (cont.)

Graphic	Octal	Numeric Decimal	Hex	Description
P	0120	80	50	Upper-case P
Q	0121	81	51	Upper-case Q
R	0122	82	52	Upper-case R
S	0123	83	53	Upper-case S
T	0124	84	54	Upper-case T
U	0125	85	55	Upper-case U
V	0126	H83	56	Upper-case V
W	0127	87	57	Upper-case W
X	0130	88	58	Upper-case X
Y	0131	89	59	Upper-case Y
Z	0132	90	5A	Upper-case Z
[	0133	91	5B	Opening (left) bracket
\	0134	92	5C	Backslash
]	0135	93	5D	Closing (right) bracket
^	0136	94	5E	Circumflex
_	0137	95	5F	Underscore
`	0140	96	60	Grave accent
a	0141	97	61	Lower-case a
b	0142	98	62	Lower-case b
c	0143	99	63	Lower-case c
d	0144	100	64	Lower-case d
e	0145	101	65	Lower-case e
f	0146	102	66	Lower-case f
g	0147	103	67	Lower-case g
h	0150	104	68	Lower-case h
i	0151	105	69	Lower-case i
j	0152	106	6A	Lower-case j
k	0153	107	6B	Lower-case k

(cont.)

Table A-1. ASCII Character Set (cont.)

Graphic	Octal	Numeric Decimal	Hex	Description
l	0154	108	6C	Lower-case l
m	0155	109	6D	Lower-case m
n	0156	110	6E	Lower-case n
o	0157	111	6F	Lower-case o
p	0160	112	70	Lower-case p
q	0161	113	71	Lower-case q
r	0162	114	72	Lower-case r
s	0163	115	73	Lower-case s
t	0164	116	74	Lower-case t
u	0165	117	75	Lower-case u
v	0166	118	76	Lower-case v
w	0167	119	77	Lower-case w
x	0170	120	78	Lower-case x
y	0171	121	79	Lower-case y
z	0172	122	7A	Lower-case z
{	0173	123	7B	Opening (left) brace
|	0174	124	7C	Vertical line
}	0175	125	7D	Closing (right) brace
~	0176	126	7E	Tilde
	0177	127	7F	Delete

Syntax of the ANSI C Language: Appendix B

This appendix contains a sorted copy of the lexical rules and the syntax for ANSI C. Lexical rules are identified by the appearance of "(LEXICAL)" in their definition.

Syntactic definitions are introduced by the name of the nonterminal symbol followed by a colon. One or more alternative definitions follow on succeeding lines.

Nonterminal symbols are printed in italic font. Terminal symbols are printed in typewriter font. Items which are optional are marked with the following subscript: *(optional)*

abstract-declarator:
    empty-abstract-declarator
    nonempty-abstract-declarator

add-op: one of
    `+ -`

additive-expression:
    multiplicative-expression
    additive-expression add-op multiplicative-expression

address-expression:
    `&` *unary-expression*

array-declarator:
    declarator `[` *constant-expression* *(optional)* `]`

assignment-expression:
    conditional-expression
    unary-expression assignment-op assignment-expression

assignment-op: one of
    `= += -= *= /= %=`
    `<<= >>= &= ^= |=`

bit-field:
    declarator *(optional)* `:` *width*

bitwise-and-expression:
    equality-expression
    bitwise-and expression `&` *equality-expression*

bitwise-negation-expression:
    ~ *unary-expression*

bitwise-or-expression:
    bitwise-xor-expression
    bitwise-or-expression | *bitwise-xor-expression*

bitwise-xor-expression:
    bitwise-and-expression
    bitwise-xor-expression ^ *bitwise-and-expression*

break-statement:
    `break ;`

case-label:
    `case` *constant-expression*

case-expression
    (*type-name*) *unary-expression*

character: (LEXICAL)
    printing-character
    escape-character

character-constant: (LEXICAL)
    ' *character* '

character-escape-code: one of (LEXICAL)
    `a n t b r f`
    `v \ ' " ?`

character-sequence: (LEXICAL)
    character
    character-sequence character

character-type-specifier:
    `char`
    `signed char`
    `unsigned char`

comma-expression:
    assignment-expression
    comma-expression , *assignment-expression*

component-declaration:
    type-specifier component-declarator-list ;

component-declarator:
    simple-component
    bit-field

component-declarator-list:
component-declarator
component-declarator-list , *component-declarator*

component-selection-expression:
direct-component-selection
indirect-component-selection

compound-statement:
{ *inner-declaration-list*$_{(optional)}$ *statement-list*$_{(optional)}$ }

conditional-expression:
logical-or-expression
logical-or-expression ? *expression* : *conditional-expression*

conditional-statement:
if-statement
if-else-statement

const-type-specifier:
`const`

constant: (LEXICAL)
integer-constant
floating-point-constant
character-constant
string-constant

constant-expression:
expression

continue-statement:
`continue ;`

decimal-constant: (LEXICAL)
nonzero-digit
decimal-constant digit

declaration:
declaration-specifiers declaration-list ;

declaration-list:
declaration
declaration-list declaration

declaration-specifiers:
storage-class-specifier
type-specifier
declaration-specifiers storage-class-specifier
declaration-specifiers type-specifier

declarator:
simple-declarator
`(` *declarator* `)`
function-declarator
array-declarator
pointer-declarator

default-label:
`default`

digit: one of (LEXICAL)
`0 1 2 3 4 5 6 7 8 9`

digit-sequence: (LEXICAL)
digit
digit digit-sequence

direct-component-selection:
postfix-expression . *identifier*

do-statement:
`do` *statement* `while (` *expression* `) ;`

dotted-digits: (LEXICAL)
digit-sequence .
digit-sequence . *digit-sequence*
. *digit-sequence*

empty-abstract-declarator:

enumeration-constant:
identifier

enumeration-constant-definition:
enumeration-constant
enumeration-constant=expression

enumeration-definition-list:
enumeration-constant-definition
enumeration-definition-list , *enumeration-constant-definition*

enumeration-tag:
identifier

enumeration-type-definition:
`enum` *enumeration-tag*$_{(optional)}$
`{` *enumeration-definition-list* `}`

enumeration-type-reference:
`enum` *enumeration-tag*
`{` *enumeration-definition-list* `}`

enumeration-type-specifier:
enumeration-type-definition
enumeration-type-reference

equality-expression:
relational-expression
equality-expression equality-op relational-expression

equality-op: one of
`==` `!=`

escape-character: (LEXICAL)
`\` *escape-code*

escape-code: (LEXICAL)
character-escape-code
numeric-escape-code

exponent: (LEXICAL)
`e` *sign-part*$_{(optional)}$ *digit-sequence*
`E` *sign-part*$_{(optional)}$ *digit-sequence*

expression:
comma-expression

expression-list:
assignment-expression
expression-list `,` *assignment-expression*

expression-statement:
expression `;`

field-list:
component-declaration
field-list component-declaration

float-marker: one of (LEXICAL)
`f F l L`

floating-constant: (LEXICAL)
digit-sequence exponent float-marker$_{(optional)}$
dotted-digits exponent$_{(optional)}$ *float-marker*$_{(optional)}$

floating-type-specifier:
`float`
`double`
`long double`

following-character: (LEXICAL)
letter
underscore
digit

for-expressions:
(*expression*$_{(optional)}$; *expression*$_{(optional)}$; *expression*$_{(optional)}$)

for-statement:
`for` *for-expressions statement*

function-body:
parameter-declaration-list $_{(optional)}$ *compound-statement*

function-call:
postfix-expression (*expression-list* $_{(optional)}$)

function-declarator:
declarator (*parameter-type-list* , . . .$_{(optional)}$)
declarator (*parameter-list*$_{(optional)}$)

function-definition:
declaration-specifiers$_{(optional)}$ *declarator function-body*

goto-statement:
`goto` *identifier* ;

hex-digit: one of (LEXICAL)
`0 1 2 3 4 5 6 7 8 9`
`A B C D E F a b c d e f`

hex-marker: one of (LEXICAL)
`0x 0X`

hexadecimal-constant: (LEXICAL)
hex-marker
hexadecimal-constant hex-digit

identifier: (LEXICAL)
underscore
letter
identifier following-character

if-else-statement:
`if` (*expression*) *statement* `else` *statement*

if-statement:
`if` (*expression*) *statement*

indirect-component-selection:
postfix-expression -> *identifier*

indirection-expression:
* *unary-expression*

initialized-declaration:
declaration-specifiers initialized-declarator-list ;

initialized-declarator:
declarator initializer-part$_{(optional)}$

initialized-declarator-list:
initialized-declarator
initialized-declarator-list , *initialized-declarator*

initializer:
expression
{ *initializer-list*$_{(optional)}$ }

initializer-list:
initializer
initializer-list , *initializer*

initializer-part:
= *initializer*

inner-declaration-list:
initialized-declaration-list

integer-constant: (LEXICAL)
decimal-constant type-marker$_{(optional)}$
octal-constant type-marker$_{(optional)}$
hexadecimal-constant type-marker$_{(optional)}$

integer-type-specifier:
signed-type-specifier
unsigned-type-specifier
character-type-specifier

iterative-statement:
while-statement
do-statement
for-statement

label:
named-label
case-label
default-label

labeled-statement:
    label : *statement*

letter: one of (LEXICAL)
```
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m
n o p q r s t u v w x y z
```

logical-and-expression:
    bitwise-or-expression
    logical-and-expression && *bitwise-or-expression*

logical-negation-expression:
    ! *unary-expression*

logical-or-expression:
    logical-and-expression
    logical-or-expression || *logical-and-expression*

long-marker: one of (LEXICAL)
```
l L
```

mult-op: one of
```
* / %
```

multiplicative-expression:
    unary-expression
    multiplicative-expression mult-op cast expression

named-label:
    identifier

nonempty-abstract-declarator:
    (*nonempty-abstract-declarator*)
    abstract-declarator ( )
    abstract-declarator [*expression*$_{(optional)}$]
    • *abstract-declarator*

nonzero-digit: one of (LEXICAL)
```
1 2 3 4 5 6 7 8 9
```

null-statement:
    ;

numeric-escape-code: (LEXICAL)
    octal-digit
    octal-digit octal-digit
    octal-digit octal-digit octal-digit
    x *hex-digit*
    x *hex-digit hex-digit*
    x *hex-digit hex-digit hex-digit*

octal-constant: (LEXICAL)
    0
    octal-constant octal-digit

octal-digit: one of (LEXICAL)
    0 1 2 3 4 5 6 7

parameter-declaration:
    declaration-specifiers declarator
    declaration-specifiers abstract-declarator

parameter-declaration-list:
    declaration-list

parameter-list:
    identifier
    parameter-list , identifier

parameter-type-list:
    parameter-declaration
    parameter-type-list , parameter-declaration

parenthesized-expression:
    (*expression*)

pointer-declarator:
    * *declarator*

pointer-declarator:
    * *type-specifier-list*$_{(optional)}$ *declarator*

postdecrement-expression:
    postfix-expression --

postfix-expression:
    primary-expression
    subscript-expression
    component-selection-expression
    function-call
    postincrement-expression
    postdecrement-expression

postincrement-expression:
    postfix-expression ++

predecrement-expression:
    -- *unary-expression*

preincrement-expression:
    ++ *unary-expression*

primary-expression:
identifier
constant
literal
parenthesized-expression

program:
top-level-declaration-list$_{(optional)}$

relational-expression:
shift-expression
relational-expression relation-op shift-expression

relational-op: one of
`< <= > >=`

return-statement:
`return` *expression*$_{(optional)}$ `;`

shift-expression:
additive-expression
shift-expression shift-op additive-expression

shift-op: one of
`<< >>`

signed-type-specifier:
`signed`
`signed`$_{(optional)}$ `int`
`signed`$_{(optional)}$ `short int`$_{(optional)}$
`signed`$_{(optional)}$ `long int`$_{(optional)}$

simple-component:
declarator

simple-declarator:
identifier

sizeof-expression:
`sizeof ( type-name )`
`sizeof` *unary-expression*

statement:
expression-statement
labeled-statement
compound-statement
conditional-statement
iterative-statement
switch-statement
break-statement
continue-statement
return-statement
goto-statement
null-statement

statement-list:
statement
statement-list statement

storage-class-specifier: one of
`auto extern register static typedef`

string-constant: (LEXICAL)
" *character-sequence*$_{(optional)}$ "

structure-tag:
identifier

structure-type-definition:
`struct` *structure-tag*$_{(optional)}$ { *field-list* }

structure-type-reference:
`struct` *structure-tag*

structure-type-specifier:
structure-type-definition
structure-type-reference

subscript-expression:
postfix-expression [*expression*]

switch-statement:
`switch` (*expression*) *statement*

top-level-declaration:
initialized-declaration
function-definition

top-level-declaration-list:
top-level-declaration
top-level-declaration-list top-level-declaration

type-marker: (LEXICAL)
    long-marker unsigned-marker$_{(optional)}$
    unsigned-marker long-marker$_{(optional)}$

type-name:
    type-specifier abstract-declarator

type-specifier:
    const-type-specifier
    enumeration type-specifier
    floating-point-type-specifier
    integer-type-specifier
    structure-type-specifier
    typedef-name
    union-type-specifier
    void-type-specifier
    volatile-type-specifier

type-specifier-list:
    type-specifier-list type-specifier

typedef-name:
    identifier

unary-expression:
    postfix-expression
    cast-expression
    sizeof-expression
    unary-minus-expression
    unary-plus-expression
    logical-negation-expression
    bitwise-negation-expression
    address-expression
    indirection-expression
    preincrement-expression
    predecrement-expression

unary-minus-expression:
    -- *unary-expression*

unary-plus-expression:
    ++ *unary-expression*

underscore: (LEXICAL)
    _

union-tag:
    identifier

union-type-definition:
union *union-tag(optional)* { *field-list* }

union-type-reference:
union *union tag:*

union-type-specifier:
union-type-definition
union-type-reference

unsigned-marker: one of (LEXICAL)
u U

unsigned-type-specifier:
unsigned short int*(optional)*
unsigned int*(optional)*
unsigned long int*(optional)*

void-type-specifier:
void

volatile-type-specifier:
volatile

while-statement:
while (*expression*) *statement*

width:
expression

Error Messages: Appendix C

Overview

This appendix lists the messages produced by Microtec Research compilers. A description of the error and an example of an error are provided whenever possible.

Note that messages are listed alphabetically by the first significant word (words like the, an, a, etc. are not considered significant). If the first word is a variable which is supplied at the time of the message, the message is indexed by the next significant word. For example, the message "*name* `undefined`" would be listed alphabetically under "undefined".

Message Severity Levels

There are two classes of errors that can occur during linker program execution. The first class is non-fatal and processing proceeds after the error is reported. The second class is fatal and processing is abandoned.

Messages are preceded with a letter in parentheses, indicating the severity type of the message. Table C-1 lists the message severity levels.

Table C-1. Message Severity Levels

Type	Severity	Meaning
(F)	Fatal	Always fatal. Processing aborts.
(E)	Error	Usually fatal, but processing continues to facilitate further error checking.
(W)	Warning	Not fatal.
(I)	Informational	Informative and often appears in conjunction with another error message.

Some messages are also preceded with the word [ANSI]. This indicates that an error was made against an ANSI-related feature.

Compiler Messages

does not appear before a parameter; ignored, (W)

The # operator does not appear before a parameter. This message is from the preprocessor.

Example:

```
#define g(b) # y
```

has no left argument; ignored, (W)

The ## operator has no left hand argument. This message is from the preprocessor.

Example:

```
#define f(a) ## a
```

has no right argument; ignored, (W)

The ## operator has no right hand argument. This message is from the preprocessor.

Example:

```
#define f(a) a ##
```

& operator has no effect, (W)

The address operator is redundant in the expression. This usually occurs whenever there is an assignment of a function pointer, since the compiler automatically assigns the address of the function.

Example:

```
int foo() {int (*p)(); f();p = &f; }
```

/* inside a comment, (I)

Comments cannot be nested. This informational message indicates that you are trying to nest comments; it is emitted when the -nQ option is set. This message is from the preprocessor.

Example:

```
/* /* inner */ */
```

address expected in initialization of *variable*, (W)

Whenever a pointer is initialized by a non-pointer value that is a character or an integer constant, the compiler generates the initializing assignment and issues this warning.

Example:

```
int *p = 0XFFFF;
```

adjacent string literal concatenation conflicts with "no ansi" option, (W)

The concatenation of adjacent strings is an ANSI feature. Your file has been compiled with the /noansi (VMS) or -nA (UNIX/DOS) option.

Example:

```
#pragma options -nA
char *str = "abc" "cdf";
```

aggregate initialization conflicts with "no ansi" option, (W)

Function prototypes are ANSI extensions. Your file has been compiled with the /noansi (VMS) or -nA (UNIX/DOS) option, but the compiler found a function with a prototype specification. Function prototypes are supported under the ANSI option.

Example:

```
#pragma options -nA
struct S { int i; int j; } s0 = {1, 2};
struct S s1 = s0;
```

[ANSI] argument is incompatible with formal parameter type, (W)

Whenever the arguments passed to a function do not match the function prototype specification (e.g., passing pointers when integers are expected and vice versa), a warning is issued to highlight the type incompatibility.

Example:

```
int foo(int i);
int baz(int *p);

int bar() { int *v; foo(v); baz(100);}
```

argument is void, (E)

The compiler attempts to apply the C type conversion rules whenever it finds function arguments that do not match the function prototype. In K&R C, no function prototypes are available, so the compiler applies default conversions to type int. However, when a function returning a void is used as an argument, the compiler cannot make any conversions; hence, the compiler generates an error that the argument is of type void.

Example:

```
void foo();
void zap();

int bar() { int i; zap(foo()); }
```

arithmetic type expected, (E)

The compiler checks to see if the expression that is evaluated contains types on which arithmetic operations can be performed. These are the C types like char, int, long, double, and float. Only addition and subtraction operations are defined for pointers. Other operations on pointers like unary operations, multiplication, division, etc. are incorrect.

Example:

```
int foo() { int *r = 0; int k; k = (int)(r*4); }
```

assembler inlining function interpreted as user function due to "no MRI extension" option, (I)

ASM directives are only supported if the Microtec Research extensions option flag is on. This is a default option. -nx compiles the file with no Microtec Research extensions.

Example:

```
#pragma options -nx
int foo() { ASM("4"); }
```

assignment to array or function, (E)

In most cases, the compiler treats an array as if it were equivalent to a pointer; however, it does not permit an assignment to an array.

Example:

```
int foo() { int a[10]; int *p; p = a; a = p; }
```

[ANSI] attempt to modify a "const", (E)

A variable declared with the const qualifier was used as an lvalue and, therefore, there was an attempt to modify the const variable.

Example:

```
int foo() { const int x; x=5; }
```

break statement is not inside a loop or switch, (E)

A break statement has appeared outside of a while, for, do, or switch statement.

Example:

```
int foo() { break; }
```

call of a nonfunction, (E)

A function name is treated as a pointer to the function of its return type. The compiler allows calls to be made only if it is a pointer to a function.

Example:

```
int foo() {void *vvar = 0; void (*vfunc)(); vfunc(); vvar();}
```

can't locate code generator

There was an error in installing the compiler.

can't open file *file_name*, (F)

The compiler cannot open the named file, either because the input file does not exist in the specified directory or, if the file exists, because the compiler encountered an access error. On UNIX systems, the error is due to an incorrect read permission on the file.

can't open #include file *file*; directive ignored, (W)

The specified include file *file* cannot be opened. This message is from the preprocessor.

Example:

```
#include "XYZ:FILE"
```

can't open output file *file*, (F)

If you invoke the preprocessor on the command line and direct the output to a file that the preprocessor cannot write to, you will get this fatal error message.

Example:

If you invoke the preprocessor, specify `/ouch.i` as the output file, and you do not have write permission to the / directory, you will get the following error message:

```
(F) can't open output file /ouch.i
```

can't open source file *file*, (F)

Self-explanatory. This message is from the preprocessor.

Example:

If you invoke the preprocessor, specify `ab.c` as the input file, and `ab.c` does not exist, you will get the following error message:

```
(F) can't open source file ab.c
```

case label is not inside a switch statement, (E)

case is a reserved keyword and can only appear within an enclosing switch statement.

Example:

```
int foo() { int i; case 0: i = 1; }
```

cast of a nonscalar type, (E)

Casts in expressions are used to inform the compiler that a type conversion is required. The compiler uses the C language rules to perform the type conversion. These conversions can only occur only on scalar types like integers, pointers, etc. A variable of struct type is not a scalar type and hence cannot be an rvalue (right hand of assignment) in a cast expression.

Example:

```
struct tag { int i; } s;
int foo() { int i; i = (int)s; }
```

cast of void type, (E)

Casts in expressions are used to inform the compiler that a type conversion is required. The compiler uses the C language rules to perform the type conversion. These conversions can occur only on scalar types like integers, pointers, etc. A void cannot be used in a cast expression.

Example:

```
int foo() { double x; x = (double)(void)0; }
```

cast to a nonscalar type, (E)

Casts in expressions are used to inform the compiler that a type conversion is required. The compiler uses the C language rules to perform the type conversion. These conversions can only occur on scalar types like integers, pointers, etc. A variable of struct type is not a scalar type and hence cannot be an lvalue (left hand of assignment) in a cast expression.

Example:

```
struct tag { int i; } s; int foo() { int i; s = (struct tag)i; }
```

[ANSI] cast used as an lvalue, (E)

Casts that would potentially change the underlying type of lvalues (the left hand of an assignment) may lead to undefined behavior. Therefore, the compiler generates a warning if your code is compiled with the strict ANSI option.

Example:

```
#pragma options -A -nx
int foo() { unsigned short ch; (signed int)ch = -1; }
```

compilation terminated: bad command line arguments, (F)

This error is caused by an internal problem with the compiler. Please contact Microtec Research Technical Support with your test case.

compilation terminated: corrupted heap, (F)

This error is caused by an internal problem with the compiler. Please contact Microtec Research Technical Support with your test case.

compilation terminated: file not found, (F)

This error is caused by an internal problem with the compiler. Please contact Microtec Research Technical Support with your test case.

compilation terminated: inconsistent data table, (F)

This error is caused by an internal problem with the compiler. Please contact Microtec Research Technical Support with your test case.

compilation terminated: interrupt occurred, (F)

This error is caused by an internal problem with the compiler. Please contact Microtec Research Technical Support with your test case.

compilation terminated: not enough memory, (F)

This error is caused by an internal problem with the compiler. Please contact Microtec Research Technical Support with your test case.

compilation terminated: parser limit exceeded, (F)

This error is caused by an internal problem with the compiler. Please contact Microtec Research Technical Support with your test case.

compilation terminated: symbol table overflow, (F)

This error is caused by an internal problem with the compiler. Please contact Microtec Research Technical Support with your test case.

compilation terminated: write error, (F)

This error is caused by an internal problem with the compiler. Please contact Microtec Research Technical Support with your test case.

compiler and input hopelessly out of sync, (F)

An unrecoverable syntax error was encountered. The error was severe enough to prevent the compiler from parsing the rest of the program. Correct the error and recompile.

constant *constant* exceeds machine capacity; truncated, (W)

Each machine places a limitation on the integer value that can be represented. If the integer specified exceeds the machine capacity, the compiler generates a warning.

Example:

```
int i = 0x100000000;
```

constant expression expected, (E)

The expression associated with a case label in a switch statement needs to be a constant expression that can be evaluated at compile time. The compiler does constant folding and evaluates the expression during compile time.

Example:

```
struct tag { int i; } s; int foo() { int i; s = (struct tag)i; }
```

constant in string *string* exceeds machine capacity; truncated, (W)

Each machine places a limitation on the integer value that can be represented. If the integer within a string exceeds the machine capacity, the compiler generates a warning.

Example:

```
char *p = "str\x100000000";
```

continue statement is not inside a loop, (E)

A **continue** statement has appeared outside of a **while**, **for**, or **do** statement.

Example:

```
int foo() { continue; }
```

default label is not inside a switch statement, (E)

default is a reserved keyword and can only appear within an enclosing **switch**.

Example:

```
int foo() { default: ; }
```

#define or #undef of *identifier* ignored, (W)

A #define or #undef statement is ignored. The preprocessor ignores the following #undefs:

__DATE__

defined

__FILE__

__LINE__

__STDC__

__TIME__

The preprocessor ignores the following `#define` macro: `defined`.

Example:

```
#define defined
```

division by 0; result assumed 0, (W)

A division by zero was attempted. This message is from the preprocessor.

Example:

```
#if 1/0
```

division by zero, (W)

Division by zero in many computer architectures results in an arithmetic exception. However, if undetected, it may lead to undefined execution behavior. The compiler tries to detect if the divisor in a divide operation is zero and, if so, it generates a warning.

Example:

```
int foo() { int i; i = 4/0; }
```

duplicate case label *number*, (E)

The constant expression associated with a case label should evaluate to unique constant integer values. Any duplication of case label expressions is reported by the compiler.

Example:

```
int foo()
{ int i;
i = bar();
switch (i) { case 1: break; case 3-2: break; }}
```

duplicate default label, (E)

There can be only one default label associated with a switch statement. Any duplication of default labels is reported by the compiler.

Example:

```
int foo()
{ int i;
i = bar();
switch (i) { default: break; default: break; }}
```

duplicate statement label *label*, (E)

Statement labels should be unique within a function scope. Any duplication of a statement label is reported by the compiler.

Example:

```
int foo()
{ int i;
i = bar();
a : if (i == 0) { a: ; } } int bar() { int r; a:; }
```

#elif used outside #if/#endif pair; ignored, (W)

An #elif statement does not occur within an #if/#endif sequence. This message is from the preprocessor.

Example:

```
#elif 1
```

#else used outside #if/#endif pair; ignored, (W)

An #else statement does not occur within an #if/#endif sequence. This message is from the preprocessor.

Example:

```
#ifdef __STDC__
      volatile int a;
#endif
#else
```

empty character literal, (W)

Empty character literal symbols were used. This message is from the preprocessor.

Example:

```
int a = '';
```

[ANSI] empty declaration, (W)

A declaration is missing an identifier. Fix the error by declaring an identifier.

Example:

```
int;
```

empty hexadecimal constant; "0x0" assumed, (W)

The hexadecimal prefix appears without an associated number. This message is from the preprocessor.

Example:

```
int a = 0x;
```

[ANSI] empty hexadecimal sequence; 'x' assumed, (W)

The escape sequence \x, etc. is reserved in ANSI; therefore, the escape sequence is ignored. This ANSI behavior can be suppressed if you compile your program with the /noansi (VMS) or -nA (UNIX/DOS) option.

Example:

```
int foo() { char p; p = '\x'; }
```

#endif does not have a matching #if; ignored, (W)

An #endif statement does not have a matching #if statement. This message is from the preprocessor.

Example:

```
#if __STDC__
      volatile int a;
#endif
#endif
```

enum value is not a constant, (E)

Only constant expressions can be used to assign enumerator values. The compiler performs constant folding and evaluates the expression during compile time.

Example:

```
int foo() { int x = 1; enum { a =0, b = x }; }
```

[ANSI] enumerated data type in bit-field declaration; accepted, (W)

An enumerator cannot be used in a bit field specification. Bit fields are specified for integers only.

Example:

```
#pragma options -nx
struct S { enum color el : 3; };
```

#error: *message*, (F)

The #error directive is used. This directive causes the ANSI C compliant preprocessor to issue a diagnostic message that includes the tokens specified in the #error directive. This message is from the preprocessor.

Example:

```
#error oops
```

[ANSI] escape sequence '*character*' reserved for future use; accepted as '*character*', (I)

The escape sequence *character*, etc. is reserved in ANSI; therefore, the escape sequence is ignored. This ANSI behavior can be suppressed if you compile your program with the /noansi (VMS) or -nA (UNIX/DOS) option.

Example:

```
int foo() { char *p; p = "\s"; }
```

extra files specified; remainder ignored, (W)

The preprocessor was invoked with an extra (third) file specified on the command line.

[ANSI] extra parameter(s) following a void prototype, (E)

This is a function prototype specification error. Only function pointers of type void can be passed as function arguments. Fix the error by including a parameter name for the void.

Example:

```
int foo(void , int );
```

file not found *file_name*

The compiler cannot find the specified file.

function *identifier* declared as *keyword*, (E)

The auto and register keywords apply to variables and specify a location. They cannot be used to specify the return type in a function declaration nor in its parameter types.

Example:

```
int foo() { auto int g(); }
```

function *function* has incomplete return type, (E)

A function can only return with a complete return type although the return statement is not explicitly stated. The compiler tries to return the correct type but finds the type has not been fully specified; hence, the error.

Example:

```
struct S foo() { }
```

function *function_name* is redefined, (E)

A function can be defined only once although it can be declared a number of times. In C, the file is the separate compilation module; therefore, only the linker can report if a function has been defined in more than one file.

Example:

```
int bar();
int bar();
int foo() { int i; i = 0; }
int foo() { int i; i = 0; }
```

Identifier *identifier* declared as void, (E)

A formal parameter was incorrectly declared to be of type **void**.

Example:

```
int foo(int x, void y);
```

Identifier expected; assumed undefined, (W)

The **#ifdef** should be supplied with an identifier as a parameter. If it is incorrectly supplied, the ifdef construct is assumed to be undefined; hence, the enclosing code between the **#ifdef** and its corresponding **#endif** is left out. This message is from the preprocessor.

Example:

```
#ifdef 1
int i;
#endif
```

#if does not have a matching #endif; supplied, (W)

A **#if** or a **#ifdef** should have a corresponding **#endif** construct. This message is from the preprocessor.

Example:

```
#if 1
int i;
```

illegal char in macro definition; symbol *symbol* deleted, (W)

A comma was expected; the unexpected symbol was deleted to correct the situation. For instance, f(a) and f(a;) both yield abc. This message is from the preprocessor.

Example:

```
#define f(a;) abc
```

incompatible type returned, (E) or (W)

The type returned is incompatible with the declared type of the function.

Example:

```
int foo() { char *p = "abc"; return p; }
struct tag { int i; }  bar() { return 0; }
```

inconsistent redeclaration of *identifier*, (E)

An identifier of a declared type cannot be redeclared to be of a different type. The compiler generates an error message whenever it detects this condition.

Example:

```
int x; char x;
```

inconsistent redeclaration of tag *tag*, (E)

A structure tag of a declared type cannot be redeclared to be a structure tag for a different structure. The compiler generates an error message whenever it detects this condition.

Example:

```
struct S { int x; }; struct S { char ch; };
```

inconsistent use of tag *tag*, (E)

A tag of a declared type cannot be used as a tag for a different type. The compiler generates an error message whenever it detects this condition.

Example:

```
struct S ; enum S;
```

increment/decrement of array or function, (E)

Pointer arithmetic operations cannot be performed on an array or function pointer.

Example:

```
int foo() { int a[10]; a++; }
```

[ANSI] inefficient evaluation in "double" because of "double" constant, (I)

Whenever double constants are used instead of integer constants, the compiler generates an informational message to indicate that the conversion is an inefficient one.

Example:

```
int foo() { float f1 = 0; f1 = f1 + 1.0; }
```

integer expected in initialization of *variable*, (W)

The initializer supplied in an initializer list should be type compatible with the variable it is expected to initialize.

Example:

```
int i = &i;
```

integer line number expected; directive ignored, (W)

A line directive must contain a reference to a line number. This message is from the preprocessor.

Example:

```
#line =
int i;
```

integral type expected, (E)

The compiler expects an integer type or a type that can be implicitly converted to an integer.

Example:

```
int foo() { int i = ~1.0; }
```

Integral value expected; real value truncated, (W)

Whenever an lvalue is of type integer, and the rvalue is of type float or of an arithmetic type which is larger than the sizeof(int), the compiler generates a warning that the rvalue is being truncated to suit the sizeof the integer. This may lead to unexpected run-time behavior.

Examples:

```
int foo() { int array[3.4]; }

int foo() { int *p; float q; p+q; }
```

[ANSI] *type* interpreted as "double", (W)

A declaration with long and float type specifiers does not increase the precision of the type. Therefore, the compiler generates a warning that these type specifiers are interpreted to be of type double.

Example:

```
long float lf;
```

Interrupt function has parameter(s), (W)

An interrupt function cannot have parameters. The compiler will generate a warning whenever it detects such a condition.

Example:

```
void interrupt foo(int i);
```

Interrupt function returns non-void type, (W)

An interrupt function must always return a void. The compiler will generate a warning whenever it detects a non-void return type.

Example:

```
int interrupt foo();
```

invalid argument for assembler inlining function, (E)

An invalid argument was used with the ASM pseudo-function.

Example:

```
f() { ASM(3); }
```

invalid array component type, (E)

An array cannot be of type void.

Example:

```
void arry[1000];
```

invalid array size, (E)

The size of an array should be a positive integer and is the result of a constant expression. The compiler evaluates the expression and checks to see that it is a positive integer. Any incorrect size specification will result in this error.

Example:

```
int arry[-1000];
```

invalid digit in octal constant *constant*; assumed decimal, (W)

In C, all octal numbers are identified by a leading zero and must contain only octal digits. This message is from the preprocessor.

Example:

```
int a = 08;
```

invalid #include file name *symbol*; directive ignored, (W)

An unacceptable file name has been chosen for an include file. This message is from the preprocessor.

Example:

```
#include 123
int i;
```

invalid initialization for *identifier*, (E)

The initialized variable requires an initializer of the same type. The compiler attempts to convert the types if a type conversion is possible. Otherwise, the compiler reports an error that the initialized variable has been supplied with an invalid initializer.

Example:

```
void (*p)() = 0;
int a[10] = p;
```

invalid insert *symbol* for assembler inlining function, (E)

An ASM instruction cannot contain references to undeclared variables in the C program. It should also be syntactically correct. If the variable is undeclared or a syntax error occurs within the ASM string, the compiler generates an error message.

Example:

```
int foo() { ASM("mov 'undef', 'truncat"); };
```

invalid return type, (E)

The function return type cannot be an array. The compiler will generate an error whenever it detects this condition.

Example:

```
typedef int A[10];
A foo();
```

invalid size for bit-field *bit-field_name*, (E)

Bit field members cannot specify a field number which is greater than the size of the type which the bit field belongs to.

Example:

```
struct S { unsigned int a : 100; unsigned int b; };

/*
struct S1 { unsigned int a : sizeof(unsigned int) + 1;
unsigned int b; };
*/
```

[ANSI] invalid specifier(s) used with "void", (E)

No addition type specifiers are used along with the type void except for const and volatile. The compiler generates an error (ANSI or otherwise) if it detects such a condition.

Example:

```
unsigned void* a;
```

invalid storage class *storage_class* for parameter *parameter*, (E)

Function parameters are formals and cannot be associated with any storage specifications.

Example:

```
int f(static int i);
```

invalid storage class *storage_class* for parameter *parameter*, (W)

An invalid automatic storage class is being specified for a formal parameter.

Example:

```
int foo(x) auto x; { }
```

invalid type for member *member_name*, (E)

The members of a structure should be a valid type. Valid types include the built-in C types and any user-defined structs, unions, or enums. Members cannot be of type void or type function.

Example:

```
struct S { struct R b; int i; };
```

identifier is not a parameter, (E)

The parameter names in a function definition should match with their declaration. The compiler detects an error if the match fails.

Example:

```
int foo(a) int b; { int k; }
```

local "extern" variable *identifier* is initialized, (E)

External variables declared in a function scope cannot be initialized. The compiler generates an error message when it detect this condition.

Example:

```
int foo() { extern int e = 0; }
```

[ANSI] local function *identifier* declared as "static", (I)

Whenever a function is declared static in a function scope, an informational message is generated.

Example:

```
int foo() { static f(); }
```

macro *identifier* invoked with wrong number of arguments, (W)

A macro has been invoked with an incorrect number of arguments. This message is from the preprocessor.

Example:

```
define f(a) a+1
void main() {
     int k = f(1,2);
     int j = f(1);
}
```

macro name *symbol* is not an identifier; directive ignored, (W)

The name given in a macro is not an identifier. This message is from the preprocessor.

Example:

```
#define 0
#define $6.0-U.S-Dollars
int j;
```

macro parameter *symbol* is not an identifier; ignored, (W)

The parameters of a macro should be identifiers. This message is issued if any of the macro parameters are not identifiers. This message is from the preprocessor.

Example:

```
#define f(0)
void main() {
     f(0);
}
```

meaningless "interrupt" qualifier for *identifier*; ignored, (W)

Although the C grammar allows a specifier list to be attached to any identifier, structure member name, function name, etc., some specifier combinations will not appear meaningful in the given context.

Example:

```
int interrupt a;
```

[ANSI] missing definition for static function *identifier*, (E) or (I)

A static function has been declared and used but not defined. This is an error when compiling with the strict ANSI option. If the declared static function is not used and the file has been compiled with the strict ANSI option, an informational message is generated.

Example:

```
#pragma options -A -nx
static foo(); int bar() { foo(); };
static zap();
```

[ANSI] missing definition for static function *identifier*; "extern" assumed, (W)

A static function was undefined; extern was assumed.

Example:

```
static foo(); int bar() { foo(); };
```

missing exponent in real constant *constant*; E0 assumed, (W)

An exponent has been left off of a real constant. This message is from the preprocessor.

Example:

```
float a = 1.E;
```

missing #include file name; directive ignored, (W)

The include file name is missing in an #include directive. This message is from the preprocessor.

Example:

```
#include
void main() {}
```

[ANSI] missing parameter name, (E)

A function definition in ANSI must contain the complete prototype specification. This error is caused by a missing parameter name.

Example:

```
int foo(int ) { }
```

[ANSI] missing type; "int" assumed, (I)

Whenever a declaration is made, the compiler expects a type specifier. However, if no type specifier is given, the compiler defaults the declaration to that of an integer and generates a warning.

Example:

```
#pragma options -nQ
a;
```

mod by 0; result assumed 0, (W)

Mod by zero was attempted. This message is from the preprocessor.

Example:

```
#if 1%0
```

module is too large for available memory, (F)

The compiler is usually very efficient about its memory allocation. However, there are certain limiting factors such as the number of macros defined, etc. The compiler generates this fatal message when it cannot allocate any more memory.

more than 40 arguments (compiler limitation), (E)

The compiler imposes a restriction on the number of function arguments to 40. This is an internal compiler limitation.

Example:

```
int foo() {
            bar( 0,0,0,0,0,0,0,0,0,0,
                 0,0,0,0,0,0,0,0,0,0,
                 0,0,0,0,0,0,0,0,0,0,
                 0,0,0,0,0,0,0,0,0,0,0
                );
          }
```

more than 8000 labels in one switch (compiler limitation), (E)

The compiler imposes a limit on the number of case labels that can be handled in a switch statement. It can process a maximum of 8000 case labels within this statement. This error message is generated when the limit has been exceeded.

name *name* specified for filler bit-field; ignored, (W)

The bit field specification lets you specify a bit field from 1 to the sizeof the type of its bit field. The bit field 0 is the filler bit field. Filler bit fields are fillers that are used to pad the bit fields to the correct size. These fields cannot be referenced. Bit field 0 is the leading filler bit field; hence, a reference to it causes the compiler to issue this warning.

Example:

```
struct S { unsigned int a : 0; };
```

no member *identifier*, (E)

Whenever a reference is made to a non-existent member of a structure, the compiler generates an error.

Example:

```
int foo() { struct S { int m; } s0; s0.p = 0; }
```

non int in bit-field declaration; accepted, (W)

Bit fields are specified for integers only. The compiler accepts some of the bit field specifications, but generates a warning.

Example:

```
#pragma options -nx
struct S { char a : 3; };
```

non-interrupt call of interrupt function, (W)

Self-explanatory.

Example:

```
interrupt void bar();
void foo() { bar(); }
```

non-prototype function declaration has formal parameter(s), (E)

A function declaration that does not have a prototype specification cannot have parameter names in its argument list. An error is generated by the compiler whenever it detects this condition.

Example:

```
int foo(a,b);
```

operand has incomplete type, (E)

A type can be used only after it has been completely specified.

Example:

```
int foo() { struct S s0; s0 = 0;}
```

operand is an address, (E)

An address is incorrectly being used as an operand; this kind of pointer arithmetic is illegal.

Example:

```
int foo() { int x,a; int *p; p=&x+&a; }
```

operand is not an lvalue, (E)

The operand is not a legal lvalue.

Example:

```
int foo() { int x; 5=x+2; }
```

operand is pointer to an object of unknown size, (E)

A pointer to an object of unknown size is incorrectly being used as an operand.

Example:

```
int foo() { struct tag *p; p++; }
```

operand is pointer to function, (E)

A pointer to a function is incorrectly being used as an operand; this kind of pointer arithmetic is illegal.

Example:

```
int foo() { int (*p) (); p++; }
```

operand of & is a bit field, (E)

A bit field is incorrectly being used as an operand of the "address of" operator (&). Fields do not have addresses, so the & operator cannot be applied to them.

Example:

```
int foo() { struct { int bit:1;} s; bar(&s.bit); }
```

operand of & is a register variable, (E)

A register variable is incorrectly being used as an operand of the "address of" operator (&). Register variables do not have addresses, so the & operator cannot be applied to them.

Example:

```
int foo() { register x; bar(&x); }
```

operand of & is not an lvalue, (E)

The operand of the "address of" operator is not legal since it is not an lvalue.

Example:

```
int foo() { int x; x=&5; }
```

operands of *operator* have incompatible types, (W)

The operands of an expression are incompatible.

Example:

```
int foo() { int x; if (&x==1) x=7; }
```

option *option1* is being phased out; use *option2*, (W)

Some options are no longer used by the compiler. Use of these options does not produce the expected behavior. The compiler will warn about the use of the obsolete option.

Example:

```
#pragma options -68332
int i;
```

[ANSI] parameter declaration outside() in a prototype definition, (E)

A parameter was declared outside of the parentheses in a function prototype declaration.

Example:

```
int foo(int a) int b; { }
```

parameter *identifier* is hidden by redeclaration, (E)

An identifier declared as a parameter has been redeclared as a local variable; this causes a conflict with respect to the retrieval of that identifier in the function scope.

Example:

```
int foo(x) { int x; bar(); }
```

pointer or array type expected, (E)

Something other than a pointer or array type variable is incorrectly being used as an operand of unary * or [ ].

Example:

```
int foo() { int x, y; x=*y; }
```

[ANSI] pointer to const becomes unrestricted, (W)

Whenever const pointers are passed as arguments to functions that require non-const pointers, there is a potential that the called function may change the object pointed to by the const pointer. Therefore, the compiler generates a warning whenever your program is compiled with the strict ANSI option.

Example:

```
#pragma options -A
int bar(int* ip);
int foo() { const int* jp; bar(jp); }
```

[ANSI] pointer to volatile becomes unrestricted, (W)

Whenever volatile pointers are passed as arguments to functions that require non-volatile pointers, there is a potential that the called function may be optimized, thereby optimizing the object pointed to by the volatile pointer. The compiler generates a warning when this condition is detected.

Example:

```
#pragma options -A
int bar(int* ip);
int foo() { volatile int* jp; bar(jp); }
```

redeclaration of *identifier*, (E)

Redeclaration of identifiers is not allowed. Two identifiers that have the same name cannot share the same scope.

Example:

```
int foo() { int x; int x; }
```

redeclaration of enum constant *constant*, (E)

An identifier appeared in an enumeration list more than once. Enumeration lists provide a convenient way to associate constant values with names, so an identifier name cannot appear twice.

Example:

```
int foo() { enum {x,y,x}; }
```

redeclaration of member *field_name*, (E)

The elements mentioned in a structure (i.e., members) cannot be declared more than once.

Example:

```
int foo() { struct x {int y; int y;}; }
```

redeclaration of parameter *identifier*, (E)

The same parameter cannot be declared more than once in a function definition.

Example:

```
int foo(int x, int x);
```

redefinition of macro *identifier*, (W)

A macro has been defined more than once. All redefinitions (e.g., the second definition and the third definition if the user supplies three definitions for the same macro name) are ignored. This message is from the preprocessor.

Example:

```
#define x y
#define x z
```

reinitialized variable *variable*

A variable has been incorrectly initialized more than once.

Example:

```
int x=1; x=2; int foo() { }
```

restriction: aggregates cannot be larger than *value*, (E)

The compiler imposes a restriction on the size of aggregates that can be allocated. This restriction arises due to architectural limitations in the target machine that are based upon the largest accessible piece of contiguous memory. The compiler does not allow an individual aggregate size to increase beyond this limitation. These limitations are fairly large, and it would be rare to reach this limit especially with 32-bit architectures.

Example:

```
int foo() {
char a[0x1fffffff] ;
}
```

scalar type expected, (E)

A variable of non-scalar type was used where one of scalar type was expected.

Example:

```
int foo() { struct {char x;} s; bar(s+1); }
```

[ANSI] "signed" and "unsigned" conflict, (E)

Two contrasting declarations (signed and unsigned) apply to this identifier; the unsigned declaration is always discarded.

Example:

```
int foo() { signed unsigned int x; }
```

size of bit-fields is undefined, (E)

The sizeof operator cannot be applied to a bit field.

Example:

```
int foo() { struct {int bit:1;} s; int x; x=sizeof(s.bit); }
```

size of functions is undefined, (E)

The sizeof operator cannot be applied to a function.

Example:

```
int foo() { typedef bar(); int x; x=sizeof(bar); }
```

size of void assumed to be 1, (W)

The sizeof operator cannot be applied to void since sizeof(void) is undefined.

Example:

```
int foo() { int x; x=sizeof(void); }
```

sizeof applied to an incomplete type, (E)

The sizeof operator was applied to an identifier of an incomplete type. This unary operator is used to compute the size of any object that has a known size; in this case, the size is unknown.

Example:

```
int foo() { struct tag s; int x; x=sizeof(s); }
```

source file contains no declarations, (W)

A source file is expected to contain at least one declaration. The compiler generates a warning if it finds a source file with no declarations.

Example:

```
int main()
{
   ;
}
```

specifier specified twice, (W)

A type specifier is incorrectly being used twice in a declaration.

Example:

```
int foo() { const int const x; }
```

specifier specified twice through indirection; ignored, (W)

An identifier has been used twice with the same type qualifier (const or volatile) in separate declarations.

Example:

```
int foo() { typedef const int INT; const INT x; }
```

specifiers *specifier1* and *specifier2* conflict, (E)

Two or more specifiers with conflicting or undefined meanings were given; the compiler does not discard any of the options and gives an error.

Example:

```
int foo() { short char x; }
```

specifiers "short" and "long" conflict, (E)

Two contrasting declarations (short and long) apply to this identifier; the short declaration is always discarded.

Example:

```
int foo() { short long int x; }
```

statement table is too large for available memory

The size of the statement label has exceeded the available memory.

storage class *storage_class* is invalid in file-scope, (E)

The auto or register keywords were used in a file-scope (external) declaration. The storage classes auto and register can only be used in a local scope, i.e., a function scope. These specifiers are not permissible on file scope variables.

Examples:

```
auto i;

register x;
```

storage classes *storage_class1* and *storage_class2* conflict, (E)

Two or more storage classes with conflicting meanings were given; the compiler does not discard any of the options and gives an error.

Example:

```
int foo() { static auto x; }
```

string too long (compiler limitation), (E)

The compiler cannot accept a string because it is too long.

struct or union type expected, (E)

Something other than a struct or union type was used with the "." operator, or something other than a pointer to a struct or union was used with the "->" operator.

Example:

```
int foo() { int x, y; x=(&y).a; }
```

switch statements nested more than 16 deep (compiler limitation), (E)

The switch statements have been nested more deeply than the compiler can accept.

Example:

```
int foo() { switch (0) switch(0) switch(0) switch(0) switch(0)
switch(0) switch(0) switch(0) switch(0) switch(0) switch(0)
switch(0) switch(0) switch(0) switch(0) switch(0) switch(0);
}
```

symbol table is too large for intermediate file, (F)

This message indicates that the size of the symbol table exceeded available memory.

syntax error; unexpected symbol: *symbol*, (E)

The compiler generates an error if correct C syntax is not followed in the file.

Example:

```
int i.
```

[ANSI] too few arguments, (E)

A function was called with too few arguments.

Example:

```
int foo(int x, int y) { int a; foo(a); }
```

[ANSI] too many arguments, (E)

A function was called with too many arguments.

Example:

```
int foo(int x) { int a,b; foo(a,b); }
```

too many files opened, (F)

The preprocessor opens at most 63 files. This message indicates that too many files are opened at the same time by the preprocessor.

too many initializers for *variable*, (E)

According to the declaration, too many initial values have been specified for a variable.

Example:

```
int foo() { int a[4] = {1,2,3,4,5,6}; }
```

type used as argument; replaced with its sizeof: *size*, (W)

An object type is incorrectly being used as an argument; it is replaced by its size.

Example:

```
int foo() { bar(int); }
```

unclosed character literal, (W)

A literal character should be quoted by two quotes. This message is emitted if one of the quotes of the literal character is missing. This message is from the preprocessor.

Example:

```
int a = ';
```

unclosed comment, (W)

The end of the source file was reached without finding a closed comment symbol */. This message is from the preprocessor.

Example:

```
int some_data;
/*
      This is an unclosed comment stream
<EOF>
```

unclosed string literal, (W)

A string literal is missing a double quote (") to close the literal string. This message is from the preprocessor.

Example:

```
void main() {
      char *str = "abc;
}
```

undeclared identifier *identifier*, (E)

An identifier was used without a prior declaration.

Example:

```
int foo() { x=5;}
```

undefined enum *identifier*, (W)

If an identifier has been declared to be a type of an undefined enumerator and it is subsequently used in an expression, the compiler generates this warning.

Example:

```
enum xx yy; int foo() { yy = 0; }
```

undefined label *label*, (E)

A label was used but not defined within a function.

Example:

```
int foo() { goto x; }
```

unexpected *symbol*; *symbol* inserted, (W)

An unexpected symbol was encountered, and the expected symbol was inserted to correct the situation. For example, if an end-of-line symbol is encountered before the) symbol while processing the ( ) expression, a) symbol will be inserted. This message is from the preprocessor.

Example:

```
#if (1
```

[ANSI] unexpected symbol *symbol*; rest of line ignored, (I)

An unexpected symbol was encountered in processing the #if/#ifdef/#include directives, etc. The unexpected symbol causes the rest of the line containing the directive to be ignored. This message is from the preprocessor.

Example:

```
#if 1)
```

unknown directive *directive*; ignored, (W)

The specified directive is not recognized. This message is from the preprocessor.

Example:

```
#stop
```

unknown option *option*, (W)

The compiler does not recognize the option entered on the command line.

unknown value "*unknown_suboption*" given for *option*, (W)

The compiler does not recognize the suboption entered on the command line.

Example:

```
-Fz          (UNIX)
/debug=off   (VMS)
```

use of "long double" conflicts with "no ansi" option; "double" assumed, (W)

The long and double specifiers conflict if the file has been compiled with the /noansi (VMS) or -nA (UNIX/DOS) option. The long double specifier is supported under the ANSI option.

Example:

```
#pragma options -nA
long double i;
```

use of prototypes conflicts with "no ansi" option, (W)

Function prototypes are ANSI extensions. Your file has been compiled with the /noansi (VMS) or -nA (UNIX/DOS) option, but the compiler found a function with a prototype specification. Function prototypes are supported under the ANSI option.

Example:

```
#pragma options -nA
int foo(void);
```

value returned from a void function, (E)

A value was illegally returned from a void function.

Example:

```
void foo() { int x; x=5; return(x); }
```

variable *variable* has unknown size, (E)

A variable with an incomplete type was declared.

Example:

```
int foo() { struct tag x; }
```

[ANSI] void pointer treated as char pointer, (W)

A pointer to void has been illegally used as an operand.

Example:

```
int foo() { void *p; p++; }
```

void type returned, (E)

A value of void type was returned from a function.

Example:

```
int foo() { void bar(); return (bar()); }
```

Index

Symbols

- operator 4-8
! operator 4-8
!= operator 4-8
% operator 4-8
& operator 3-3, 4-8
&& operator 4-8
* operator 4-8
+ operator 4-8
/ operator 4-8
< operator 4-8
<= operator 4-8
== operator 4-8
> operator 4-8
>= operator 4-8
?: operator 4-8
^ operator 4-8
| operator 4-8
|| operator 4-8
~ operator 4-8
_68000 preprocessor symbol 4-4
_68010 preprocessor symbol 4-4
_68020 preprocessor symbol 4-4
_68030 preprocessor symbol 4-4
_68040 preprocessor symbol 4-4

A

abort function 5-23
abs function 5-24
Absolute addressing 8-32, 8-33
acos function 5-25
"address of" operator 3-3
Addressing
    absolute 8-32
    changing the default 8-54
    direct memory 8-51
    I/O port 8-51
    PC-relative 8-35
    register-relative 8-32
Algebraic simplification optimization 9-1
Alignment
    aggregates 6-22
        array 6-22
        structures 6-22
        unions 6-22
    bit fields 6-21
Alignment of struct/union in parameter area 7-2
Allocation of data types 3-2
ANSI C syntax B-1
_APOLLO preprocessor symbol 4-3
Arithmetic conversions 3-15
Arithmetic functions
    acos 5-6
    acosf 5-8
    asin 5-6
    asinf 5-8
    atan 5-6
    atan2 5-6
    atan2f 5-8
    atanf 5-8
    ceil 5-6
    ceilf 5-8
    cos 5-6
    cosf 5-8
    cosh 5-6
    coshf 5-8
    exp 5-6
    expf 5-8
    fabs 5-6
    fabsf 5-8
    floor 5-6
    floorf 5-8
    fmod 5-6
    fmodf 5-8
    frexp 5-6
    frexpf 5-8
    ldexp 5-6
    ldexpf 5-8

log 5-6
log10 5-6
log10f 5-8
logf 5-8
modf 5-6
modff 5-8
pow 5-6
powf 5-8
sin 5-6
sinf 5-8
sinh 5-6
sinhf 5-8
sqrt 5-6
sqrtf 5-8
tan 5-6
tanf 5-8
tanh 5-6
tanhf 5-8
Arithmetic operators 3-11
Array operator synthesis optimization 9-12
Array operators 3-13
array type 3-2, 6-3
Arrays 6-17
ASCII character set A-1
asctime function 5-21, 5-26
asin function 5-27
ASM pseudo-function 8-1
asm pseudo-function 8-1
use with volatile 8-3
Assembler inlining 8-1
considerations 8-9
guidelines 8-9
Assembler source file, production of 10-1
Assembly code, optimizing by hand 10-1
Assembly language
example of a routine 7-8
interface 7-1, 7-7
assert macro 5-28
assert.h file 5-1, 5-3
Assignment operators 3-12
Assignments 3-10
atan function 5-29
atan2 function 5-30
atexit function 5-21, 5-31
atof function 5-32
atoi function 5-33
atol function 5-34
auto keyword 2-5, 3-4
Auto storage class 3-5

B

Back quote
use in variable names 8-4
_BCS preprocessor symbol 4-3
Binary expressions 3-10
Bit fields 6-14
alignment 6-21
packing 6-21
Bitwise operators 3-11
Branch tail merging 9-6
break
keyword 2-5, 3-19
statement 3-20
bsearch function 5-35
BUFSIZ 5-16, 5-141

C

C calling conventions 7-1
C language elements 2-1
calloc function 5-21, 5-36
case keyword 2-5
Casting, type 3-16
ceil function 5-37
char keyword 2-5
_CHAR_SIGNED preprocessor symbol 4-4
char type 3-2, 6-3, 6-4
_CHAR_UNSIGNED preprocessor symbol 4-4
Character handling functions
isalnum 5-4
isalpha 5-4
iscntrl 5-4
isdigit 5-4
isgraph 5-4
islower 5-4

isprint 5-4
ispunct 5-4
isspace 5-4
isupper 5-4
isxdigit 5-4
tolower 5-4
toupper 5-4
Character string constants 2-4
clearerr function 5-38
clock function 5-39
close function 5-40
Code
allocation 8-53
position-dependent 8-27
position-independent 8-27
sections 10-3
Code hoisting 9-13
/code=pc option (VMS) 8-27, 8-28
Comments 2-6
Common subexpression elimination 9-7
Compiler output listing 10-3
Complex data types 3-2, 6-3
array 6-3
struct 6-3
union 6-3
Compound statement 3-20
Conditional compilation directives 4-6
#elif 4-6
#else 4-6
#endif 4-6
#if 4-6
#ifdef 4-6
#ifndef 4-6
Conditional expressions 3-10
Configurations, memory 8-21, 8-23, 8-26, 8-47, 8-50
const
keyword 2-5
type 3-18
Constant expressions 3-7
Constant folding optimization 9-1
Constants 2-1
character string 2-4
floating-point 2-2
integer 2-1
numeric 2-1
continue
keyword 2-5, 3-19
statement 3-20
Conventions, notational vii
Conversion between strings and numbers 5-17
Conversions, usual arithmetic 3-15
cos function 5-41
cosh function 5-42
_CPU32 preprocessor symbol 4-4
creat function 5-43
Cross-jump optimization 9-14
ctime function 5-21, 5-44
ctype.h file 5-1, 5-3

D

Data
allocation 8-53
formats 6-1
initialization 8-52
position-independent 8-27
sections 10-3
shared 8-39
system 8-40
Data types
allocation 3-2
char 6-3
complex 3-2
double 3-2, 6-3, 6-9, 6-12
enum 6-3
float 3-2, 6-3, 6-9, 6-10
int 6-3
long double 6-3
long int 6-3
pointer 3-3, 6-3
range of values 6-3
scalar 3-2, 6-4
short int 6-3
size 6-3
unsigned char 6-3
unsigned int 6-3

unsigned long int 6-3
unsigned short int 6-3
void 3-3
/data=An option (VMS) 8-27, 8-28
/data=pc option (VMS) 8-27, 8-28
__DATE__ preprocessor symbol 4-1
Dead code elimination optimization 9-3
/debug=lines option (VMS) 10-3
_DEC_STATION preprocessor symbol 4-3
Declarations
function 3-26
global 3-4
local 3-4
variable 3-17
default keyword 2-5, 3-23
#define directive 4-5, 4-6, 4-9, 4-11, 8-4
defined operator 4-8
Diagnostic macros
assert 5-3
difftime function 5-45
Direct memory addressing 8-51
div function 5-46
do keyword 2-5, 3-19
double keyword 2-5
double type 3-2, 6-3
Double-precision format 6-12
do-while statement 3-20

E

EDOM error macro 5-4
#elif directive 4-6, 4-8, 4-12
#else directive 4-6, 4-13
else keyword 2-5, 3-19
Embedded systems 8-1
considerations 8-10
user-modified routines 8-14
#endif directive 4-6, 4-14
enum
keyword 2-5
type 3-2, 6-3
Enumeration types 6-16
EOF 5-16
Epilogue, function 7-7
generating code for epilogue 9-16
eprintf function 5-47
ERANGE error macro 5-4
errno error macro 5-4, 5-21
errno.h file 5-1, 5-4
#error directive 4-7, 4-15
Error macros 5-4
EDOM 5-4
ERANGE 5-4
errno 5-4
HUGE_VAL 5-7, 5-9
Error messages C-1
Escape sequences 2-4
_exit function 5-49
exit function 5-48
exp function 5-50
Expression statement 3-19
Expressions 3-6
binary 3-10
conditional 3-10
constant 3-7
primary 3-7
unary 3-8
extern keyword 2-5, 3-4
Extern storage class 3-5

F

fabs function 5-51
Factorization optimization 9-3
fclose function 5-52
Features 1-1
feof function 5-53
ferror function 5-54
fflush function 5-55
fgetc function 5-56
fgetpos function 5-57
fgets function 5-58
FILE 5-16
__FILE__ preprocessor symbol 4-1, 4-20
FILENAME_MAX 5-16
fileno macro 5-59

float
keyword 2-5
type 3-2, 6-3, 6-10
float.h file 5-1, 5-5
Floating-point
constants 2-2
data types 6-9
format 6-10
removing unneeded support 8-10
representation 3-2, 6-9
return values 7-3
type macros 5-5
fmod function 5-61
fopen function 5-62
FOPEN_MAX 5-16
for
keyword 2-5, 3-19
statement 3-21
Format conversions — see also sscanf, sprintf 5-17
Formatted I/O functions
fprintf 5-15
fscanf 5-15
printf 5-15
scanf 5-15
sprintf 5-15
sscanf 5-15
fprintf function 5-64
_FPU preprocessor symbol 4-4
fputc function 5-65
fputs function 5-66
Frame pointer 7-4
/frames option (VMS) 7-5, 9-16
fread function 5-67
free function 5-21, 5-68
frexp function 5-69
fscanf function 5-70
fseek function 5-71
fsetpos function 5-72
ftell function 5-73
ftoa function 5-74
Functions 3-25
calling at an absolute address 8-51
declaration 3-26
old-style 3-30
declaration with body 3-27
epilogue 7-7
include libraries 3-30
library
(see Library functions)
non-reentrant 5-21
prologue 7-6
local variables in prologue 7-6
prototype arguments 3-25
prototypes 3-25
reentrant 5-21
return types 3-25
return values
floating-point 7-3
integer 7-3
structure 7-3
union 7-3
system 8-10
undeclared 3-28
using prototype function declarations 3-29
fwrite function 5-75

G

General optimizations 9-1
General utility functions
abort 5-17
abs 5-17
atexit 5-17
atof 5-17
atoi 5-17
atol 5-17
bsearch 5-17
calloc 5-17
div 5-17
exit 5-17
free 5-17
labs 5-17
ldiv 5-17
malloc 5-17
qsort 5-17
rand 5-17

realloc 5-17
srand 5-17
strtod 5-17
strtol 5-17
strtoul 5-17
Generating code for function prologue and epilogue 9-16
getc function 5-76
getchar function 5-77
getl function 5-78
gets function 5-79
getw function 5-80
-Gl option (UNIX/DOS) 10-3
Global
declarations 3-4
optimizations 9-3
variables
and inserts 8-5
Global constant propagation optimization 9-5
Global copy propagation 9-5
gmtime function 5-21, 5-81
goto
keyword 2-5, 3-19
statement 3-22
Grouping stack adjust instructions optimization 9-16

H

-h option (UNIX/DOS) 8-11, 8-12, 8-24
HP 64000 systems 8-11
requirements 8-13
section names 8-12
use with ASM68K Assembler 8-12
use with MCC68K Compiler 8-11
user-modified routines 8-17
/hp64000 option (VMS) 8-11, 8-12, 8-24
_HP9000_300 preprocessor symbol 4-3
_HP9000_700 preprocessor symbol 4-3
_HP9000_800 preprocessor symbol 4-3
HUGE_VAL error macro 5-7, 5-9

I

I/O functions 5-13
clearerr 5-14
fclose 5-14
feof 5-14
ferror 5-14
fflush 5-14
fgetc 5-14
fgetpos 5-14
fgets 5-14
fopen 5-14
fputc 5-14
fputs 5-14
fread 5-14
fseek 5-14
fsetpos 5-14
ftell 5-14
fwrite 5-14
getc 5-14
getchar 5-14
gets 5-14
perror 5-14
putc 5-14
putchar 5-14
puts 5-14
remove 5-14
rewind 5-14
setbuf 5-14
ungetc 5-14
vfprintf 5-15
vprintf 5-15
vsprintf 5-15
I/O port addressing 8-51
Identifiers 2-1
IEEE floating-point format 6-9
if
keyword 2-5, 3-19
statement 3-22
#if directive 4-6, 4-8, 4-16
#ifdef directive 4-6, 4-9, 4-17
if-else statement 3-22
#ifndef directive 4-6, 4-9, 4-18
#include directive 3-6, 4-4, 4-9, 4-19, 5-1

Include files 5-1
assert.h 5-1
ctype.h 5-1
errno.h 5-1
float.h 5-1
limits.h 5-1
locale.h 5-1
math.h 5-1
mathf.h 5-1
mriext.h 5-1
setjmp.h 5-1
signal.h 5-1
stdarg.h 5-1
stddef.h 5-1
stdio.h 5-1
stdlib.h 5-1
string.h 5-1
time.h 5-1
Indexing array optimization 9-17
Induction variable elimination 9-10
INITDATA linker command 8-19, 8-52
Initializations 3-18, 8-52
compile-time 8-52
data 8-52
run-time 8-52
static variables 6-25
In-line assembly 8-1
Input/Output functions 5-13
Inserts 8-5
int
keyword 2-5
type 3-2, 6-3, 6-6
Integer return values 7-3
Integral promotion 3-15
Internal compiler errors C-8, C-9
Interrupt handlers, declaring 7-10
interrupt keyword 2-5, 7-10
Introduction 1-1
/ireturn=subroutine option (VMS) 7-10
isalnum function 5-82
isalpha function 5-83
isascii function 5-84
iscntrl function 5-85
isdigit function 5-86
isgraph function 5-87
islower function 5-88
isprint function 5-89
isspace function 5-91
isupper function 5-92
isxdigit function 5-93
itoa function 5-94
itostr function 5-95

J

Jump optimizations 9-12

K

Keywords 2-5
auto 2-5
break 2-5
case 2-5
char 2-5
const 2-5
continue 2-5
default 2-5
do 2-5
double 2-5
else 2-5
enum 2-5
extern 2-5
float 2-5
for 2-5
goto 2-5
if 2-5
int 2-5
interrupt 2-5
long 2-5
packed 2-5
register 2-5
return 2-5
short 2-5
signed 2-5
sizeof 2-5
static 2-5
struct 2-5

switch 2-5
typedef 2-5
union 2-5
unpacked 2-5
unsigned 2-5
void 2-5
volatile 2-5
while 2-5
-Kf option (UNIX/DOS) 7-5, 9-16
-Kr option (UNIX/DOS) 7-10
-Ku option (UNIX/DOS) 6-4

L

L_tmpnam 5-16
Labeled statement 3-22
labs function 5-96
ldexp function 5-97
ldiv function 5-98
Libraries 8-18
Library functions 5-1
_exit 5-49
_tolower 5-175
_toupper 5-177
abort 5-23
abs 5-24
acos 5-25
asctime 5-26
asin 5-27
atan 5-29
atan2 5-30
atexit 5-31
atof 5-32
atoi 5-33
atol 5-34
bsearch 5-35
calloc 5-36
ceil 5-37
clearerr 5-38
clock 5-39
close 5-40
cos 5-41
cosh 5-42
creat 5-43
ctime 5-44
difftime 5-45
div 5-46
eprintf 5-47
exit 5-48
exp 5-50
fabs 5-51
fclose 5-52
feof 5-53
ferror 5-54
fflush 5-55
fgetc 5-56
fgetpos 5-57
fgets 5-58
floor function 5-60
fmod 5-61
fopen 5-62
fprintf 5-64
fputc 5-65
fputs 5-66
fread 5-67
free 5-68
frexp 5-69
fscanf 5-70
fseek 5-71
fsetpos 5-72
ftell 5-73
ftoa 5-74
fwrite 5-75
getc 5-76
getchar 5-77
getl 5-78
gets 5-79
getw 5-80
gmtime 5-81
isalnum 5-82
isalpha 5-83
isascii 5-84
iscntrl 5-85
isdigit 5-86
isgraph 5-87
islower 5-88
isprint 5-89

ispunct 5-90
isspace 5-91
isupper 5-92
isxdigit 5-93
itoa 5-94
itostr 5-95
labs 5-96
ldexp 5-97
ldiv 5-98
localtime 5-99
log 5-100
log10 5-101
longjmp 5-102
lseek 5-103
ltoa 5-104
ltostr 5-105
malloc 5-106, 5-141
memccpy 5-108
memchr 5-109
memclr 5-110
memcmp 5-111
memcpy 5-112
memmove 5-113
memset 5-114
modf 5-116
open 5-118
perror 5-120
pow 5-121
printf 5-122
putc 5-126
putchar 5-127
putl 5-128
puts 5-129
putw 5-130
qsort 5-131
rand 5-132
read 5-133
realloc 5-134
remove 5-135
rewind 5-136
sbrk 5-137
scanf 5-138
setbuf 5-141
setjmp 5-142
setlocale 5-143
sin 5-144
sinh 5-145
sprintf 5-146
sqrt 5-147
srand 5-148
sscanf 5-149
strcat 5-150
strchr 5-151
strcmp 5-152
strcoll 5-153
strcpy 5-154
strcspn 5-155
strerror 5-156
strlen 5-157
strncat 5-158
strncmp 5-159
strncpy 5-160
strpbrk 5-161
strrchr 5-162
strspn 5-163
strstr 5-164
strtod 5-165
strtok 5-166
strtol 5-167
strtoul 5-168
swab 5-169
tan 5-170
tanh 5-171
time 5-172
toascii 5-173
tolower 5-174
toupper 5-176
ungetc 5-178
unimplemented 5-2
unlink 5-179
vfprintf 5-184
vprintf 5-185
vsprintf 5-186
write 5-187
zalloc 5-188

Library include files
assert.h 5-3
ctype.h 5-3
errno.h 5-4
float.h 5-5
limits.h 5-5
locale.h 5-5
math.h 5-5
mathf.h 5-7
mriext.h 5-9
setjmp.h 5-11
signal.h 5-11
stdarg.h 5-12
stddef.h 5-12
stdio.h 5-13
stdlib.h 5-17
string.h 5-18
time.h 5-19
Library macros
assert 5-28
fileno 5-59
max 5-107
min 5-115
offsetof 5-117
va_arg 5-180
va_end 5-182
va_start 5-183
Limits macros 5-5
limits.h file 5-1, 5-5
#line directive 4-1, 4-4, 4-8, 4-9, 4-20
Line number control directive
#line 4-4
Line numbers in assembly language source file 10-3
__LINE__ preprocessor symbol 4-1, 4-20
Linker
command example
HP 64000 system, IEEE 8-24
IEEE 8-20
ROM-based system, IEEE 8-22
use 8-18
Listing, compiler output 10-3
Literals, string 2-4
Local
declarations 3-4
optimizations 9-7
variables 6-25
Local constant propagation 9-8
Local copy propagation 9-9
locale.h file 5-1, 5-5
Localization functions
setlocale 5-5
localtime function 5-21, 5-99
log function 5-100
log10 function 5-101
Logical operators 3-11
long
keyword 2-5
type 3-2
Long double precision format 6-12
long double type 3-2, 6-3
long int type 6-3, 6-8
longjmp function 5-102
Loop invariant code optimization 9-10
Loop optimizations 9-9
Loop rotation optimization 9-11
lseek function 5-103
ltoa function 5-104
ltostr function 5-105

M

_M68 preprocessor symbol 4-3
m68k preprocessor symbol 4-2
Machine-dependent optimizations 9-16
Macro definition directives 4-5
#define 4-5
#undef 4-5
malloc function 5-21, 5-106
math.h file 5-1, 5-5, 5-7
Mathematical functions
acos 5-6
acosf 5-8
asin 5-6
asinf 5-8
atan 5-6
atan2 5-6

atan2f 5-8
atanf 5-8
ceil 5-6
ceilf 5-8
cos 5-6
cosf 5-8
cosh 5-6
coshf 5-8
exp 5-6
expf 5-8
fabs 5-6
fabsf 5-8
floor 5-6
floorf 5-8
fmod 5-6
fmodf 5-8
frexp 5-6
frexpf 5-8
ldexp 5-6
ldexpf 5-8
log 5-6
log10 5-6
log10f 5-8
logf 5-8
modf 5-6
modff 5-8
pow 5-6
powf 5-8
sin 5-6
sinf 5-8
sinh 5-6
sinhf 5-8
sqrt 5-6
sqrtf 5-8
tan 5-6
tanf 5-8
tanh 5-6
tanhf 5-8
mathf.h file 5-1, 5-9
max macro 5-107
MC68000 preprocessor symbol 4-2
mc68000 preprocessor symbol 4-2
_MCC68K preprocessor symbol 4-2
MCC68K preprocessor symbol 4-3
mcc68k preprocessor symbol 4-3
-Mcp option (UNIX/DOS) 8-27, 8-28
-Mdn option (UNIX/DOS) 8-27, 8-28
-Mdp option (UNIX/DOS) 8-27, 8-28
memccpy function 5-108
memchr function 5-109
memclr function 5-110
memcmp function 5-111
memcpy function 5-112
memmove function 5-113
Memory configurations 8-21, 8-23, 8-26, 8-47, 8-50
Memory functions
memchr 5-19
memcmp 5-19
memcpy 5-19
memmove 5-19
memset 5-19
Memory layout 6-26
memset function 5-114
Message severity levels C-1
Messages, error C-1
Microtec Research extensions
functions 5-9
_tolower 5-10
_toupper 5-10
eprintf 5-10
fileno 5-10
ftoa 5-10
getl 5-10
getw 5-10
isascii 5-10
itoa 5-10
itostr 5-10
ltoa 5-10
ltostr 5-10
max 5-10
memccpy 5-10
memclr 5-10
min 5-10
putl 5-10
putw 5-10

swab 5-10
toascii 5-10
zalloc 5-10
macros
BLKSIZE 5-11
FALSE 5-11
NULLPTR 5-11
stdaux 5-11
stdprn 5-11
TRUE 5-11
min macro 5-115
Miscellaneous operators 3-13
Mixed operands 3-15
modf function 5-116
-Mr option (UNIX/DOS) 7-4
_MRI_EXTENSIONS preprocessor symbol 4-4
_MRI preprocessor symbol 4-2
MRI preprocessor symbol 4-3
mri preprocessor symbol 4-3
mriext.h file 5-1, 5-9
-Mrn option (UNIX/DOS) 8-37
_MSDOS preprocessor symbol 4-3
Multiple jump optimization 9-13
Multitasking environment (MTE) 8-37, 8-44
Multi-threaded environments 8-44

N

Names, section 8-12
-nOc option (UNIX/DOS) 7-5, 9-16
Non-local jump
functions
longjmp 5-11
setjmp 5-11
type
jmp_buf 5-11
Non-reentrant functions 3-6, 5-21
asctime 5-21
atexit 5-21
calloc 5-21
ctime 5-21
free 5-21
gmtime 5-21
localtime 5-21
malloc 5-21
rand 5-21
realloc 5-21
srand 5-21
strtok 5-21
zalloc 5-21
Normalized form 6-9
Notational conventions vii
NULL 5-16
NULLPTR 5-141
Numeric constants 2-1

O

offsetof macro 5-117
-Og option (UNIX/DOS) 9-12
Old-style function declaration 3-30
open function 5-118
Operands, mixed 3-15
Operators 2-5, 3-10
- 4-8
! 4-8
!= 4-8
% 4-8
& 4-8
&& 4-8
* 4-8
+ 4-8
/ 4-8
< 4-8
<= 4-8
== 4-8
> 4-8
>= 4-8
?: 4-8
^ 4-8
| 4-8
|| 4-8
~ 4-8
arithmetic 3-11
array 3-13
assignment 3-12
bitwise 3-11

defined 4-8
evaluation order of operators 3-13
logical 3-11
miscellaneous 3-13
pointer 3-13
precedence of operators 3-13
relational 3-12
structure 3-13
union 3-13
Optimizations
code hoisting 9-13
cross-jump 9-14
short/long displacement 9-14
general 9-1
algebraic simplification 9-1
constant folding 9-1
redundant code elimination 9-2
strength reduction 9-1
unreachable code 9-2
global 9-3
branch tail merging 9-6
dead code elimination 9-3
factorization 9-3
global constant propagation 9-5
global copy propagation 9-5
register coloring 9-3
unused definition elimination 9-4
jump 9-12
multiple jump 9-13
redundant jump elimination 9-13
local 9-7
common subexpression elimination 9-7
generating code for the switch statement 9-7
local constant propagation 9-8
local copy propagation 9-9
redundant load and store elimination 9-7
loop 9-9
array operator synthesis 9-12
loop invariant code optimization 9-10
loop rotation 9-11
strength reduction and index simplification 9-10
machine-dependent 9-16
generating code for function prologue and epilogue 9-16
grouping stack adjust instructions 9-16
indexing arrays 9-17
/optimize=globalflow option (VMS) 9-12
/optimize=nocombine_pops option (VMS) 7-5, 9-16
Order of evaluation of operators 3-13
Output listing, compiler 10-3

P

packed keyword 2-5, 6-23
Packed structures 6-23
packed type 6-24
Packing
bit fields 6-21
Parameters
passing 7-1
popping 7-4
setting 7-1
_PC preprocessor symbol 4-3
PC-relative addressing 8-35
perror function 5-120
_PIC preprocessor symbol 4-4
_PID preprocessor symbol 4-4
Pointer operators 3-13
pointer type 3-2, 6-3
Pointers 3-3, 6-13
Position-dependent code 8-27
Position-independent
code 8-27
data 8-27
pow function 5-121
#pragma asm directive 4-21, 8-6
#pragma directives 4-7, 4-9
#pragma endasm directive 4-21, 8-6
#pragma list directive 4-22
#pragma option directive 4-9, 4-23
Precedence of operators 3-13

Predefined symbols 4-1
__DATE__ 4-1
__FILE__ 4-1
__LINE__ 4-1
__STDC__ 4-1
__TIME__ 4-2
_68000 4-4
_68010 4-4
_68020 4-4
_68030 4-4
_68040 4-4
_APOLLO 4-3
_BCS 4-3
_CHAR_SIGNED 4-4
_CHAR_UNSIGNED 4-4
_CPU32 4-4
_DEC_STATION 4-3
_FPU 4-4
_HP9000_300 4-3
_HP9000_700 4-3
_HP9000_800 4-3
_M68 4-3
_MCC68K 4-2
_MRI 4-2
_MRI_EXTENSIONS 4-4
_MSDOS 4-3
_PC 4-3
_PIC 4-4
_PID 4-4
_SCO 4-3
_SIZEOF_CHAR 4-2
_SIZEOF_DOUBLE 4-2
_SIZEOF_FLOAT 4-2
_SIZEOF_INT 4-2
_SIZEOF_LONG 4-2
_SIZEOF_LONG_DOUBLE 4-2
_SIZEOF_POINTER 4-2
_SIZEOF_SHORT 4-2
_sun 4-3
_SUN3 4-3
_SUN4 4-3
_UNIX 4-3
_VAX 4-3
_VMS 4-3
m68k 4-2
MC68000 4-2
mc68000 4-2
MCC68K 4-3
mcc68k 4-3
MRI 4-3
mri 4-3
/prepend=dot option (VMS) 8-5
Preprocessor 4-1
Preprocessor directives
#define 4-7, 4-11
#elif 4-7, 4-12
#else 4-7, 4-13
#endif 4-7, 4-14
#error 4-7, 4-15
#if 4-7, 4-16
#ifdef 4-7, 4-17
#ifndef 4-7, 4-18
#include 4-7, 4-19
#line 4-7, 4-20
#pragma 4-7
#pragma asm 4-21
#pragma endasm 4-21
#pragma list 4-22
#pragma option 4-23
#undef 4-7, 4-24
types 4-4
#pragma 4-7
conditional compilation 4-6
line number control 4-4
macro definition 4-5
source file include 4-4
Primary expressions 3-7
printf function 5-122
Program termination 5-48, 5-49
Prologue, function 7-6
generating code for prologue 9-16
Promotions, integral 3-15
Prototypes, function 3-25
arguments 3-25
declaration
use 3-29

Public storage class 3-5
putc function 5-126
putchar function 5-127
putl function 5-128
puts function 5-129
putw function 5-130

Q

qsort function 5-131
Questions vii
Quotation marks 2-4

R

rand function 5-21, 5-132
read function 5-133
Reader's response sheet vii
Read-only memory (ROM) 8-1
realloc function 5-21, 5-134
Redundant code elimination optimization 9-2
Redundant jump elimination optimization 9-13
Redundant load and store elimination 9-7
Reentrant code, generating 8-42
Reentrant functions 5-21, 8-42
References vi
Register
    reserving a register 8-37
    reserving for special purposes 8-41
    storage class 3-5
    use 7-4
Register coloring optimization 9-3
register keyword 2-5, 3-4
Register-relative addressing 8-32
Relational operators 3-12
remove function 5-135
/reserve option (VMS) 7-4, 8-37
return
    keyword 2-5, 3-19
    statement 3-23
Return types for functions 3-25
Returning a typed value 8-3
rewind function 5-136
ROM 8-1
ROM-based systems 8-22
Routines, user-modified 8-14, 8-17
Run-time library 5-1

S

sbrk function 5-137
Scalar data types 3-2, 6-4
    char 6-4
    int 6-6
    long int 6-8
    short int 6-5
    unsigned char 6-4
    unsigned int 6-7
    unsigned long int 6-8
    unsigned short int 6-6
scanf function 5-138
_SCO preprocessor symbol 4-3
Section names 8-12
    default 8-18
        changing the default 8-53
Sections
    code 10-3
    data 10-3
SEEK_CUR 5-16
SEEK_END 5-16
SEEK_SET 5-16
setbuf function 5-141
setjmp function 5-142
setjmp.h file 5-1, 5-11
setlocale function 5-143
Severity of errors C-1
Shared
    data 8-39
    program 8-38
Sharing variables 3-6
short
    keyword 2-5
    type 3-2
short int type 6-3, 6-5
Short integers 7-1
Short/long displacement optimization 9-14
Short-circuit evaluation 3-10
signal.h file 5-1, 5-11

signed keyword 2-5
_simulated_input variable 5-20
_simulated_output variable 5-20
sin function 5-144
Single quotation marks 2-4
Single-precision format 6-10
sinh function 5-145
Size of aggregates 6-22
_SIZEOF_CHAR preprocessor symbol 4-2
_SIZEOF_DOUBLE preprocessor symbol 4-2
_SIZEOF_FLOAT preprocessor symbol 4-2
_SIZEOF_INT preprocessor symbol 4-2
sizeof keyword 2-5
_SIZEOF_LONG_DOUBLE preprocessor symbol 4-2
_SIZEOF_LONG preprocessor symbol 4-2
_SIZEOF_POINTER preprocessor symbol 4-2
_SIZEOF_SHORT preprocessor symbol 4-2
Sizes of data types 6-3
Software performance report vii
Source file include directive
  #include 4-4
sprintf function 5-146
sqrt function 5-147
srand function 5-21, 5-148
sscanf function 5-149
Stack
  frames 7-5
  pointer 7-4
Standard definition
  macros
    NULL 5-13
    offset 5-13
  types
    ptrdiff_t 5-12
    size_t 5-12
    wchar_t 5-12
Statements 3-19
  break 3-20
  compound 3-20
  continue 3-20
  do-while 3-20
  expression 3-19
  for 3-21
  goto 3-22
  if 3-22
  if-else 3-22
  labeled 3-22
  return 3-23
  switch 3-23
  while 3-24
Static
  storage class 3-6
  variable names 10-2
  variables 6-25
static keyword 2-5, 3-4
stdarg.h file 5-1, 5-12
__STDC__ preprocessor symbol 4-1
stddef.h file 5-1, 5-12
stderr 5-16
stdin 5-15, 5-16
stdio.h file 5-1, 5-13
stdlib.h file 5-1, 5-17
stdout 5-15, 5-16
Storage class 3-4
  keywords 3-4
    auto 3-4
    extern 3-4
    register 3-4
    static 3-4
    typedef 3-4
Storage layout 6-1
strcat function 5-150
strchr function 5-151
strcmp function 5-152
strcoll function 5-153
strcpy function 5-154
strcspn function 5-155
Strength reduction and index simplification 9-10
Strength reduction optimization 9-1
strerror function 5-156
String functions 5-18
  strcat 5-18
  strchr 5-18
  strcmp 5-18

strcoll 5-18
strcpy 5-18
strcspn 5-18
strerror 5-18
strlen 5-18
strncat 5-18
strncmp 5-18
strncpy 5-18
strpbrk 5-18
strrchr 5-18
strspn 5-18
strstr 5-18
strtok 5-18
String literals 2-4
string.h file 5-1, 5-18
Strings 6-18
strlen function 5-157
strncat function 5-158
strncmp function 5-159
strncpy function 5-160
strpbrk function 5-161
strrchr function 5-162
strspn function 5-163
strstr function 5-164
strtod function 5-165
strtok function 5-21, 5-166
strtol function 5-167
strtoul function 5-168
struct
keyword 2-5
type 3-2, 6-3
struct/union in parameter area, alignment of 7-2
Structure
operators 3-13
return value 7-3
Structures 6-18
packed 6-23
Suggestions vii
_sun preprocessor symbol 4-3
_SUN3 preprocessor symbol 4-3
_SUN4 preprocessor symbol 4-3
swab function 5-169
switch
keyword 2-5, 3-19
statement 3-23
switch statements, generating code for 9-7
Syntax, ANSI C B-1
System data 8-40
System functions 5-20, 8-10
_exit 5-20
close 5-20
creat 5-20
lseek 5-20
open 5-20
read 5-20
sbrk 5-20
unlink 5-20
write 5-20

T

tan function 5-170
tanh function 5-171
Threads
definition 8-42
multiple 8-42
Time and date functions 5-19
asctime 5-19
clock 5-19
ctime 5-19
difftime 5-19
gmtime 5-19
localtime 5-19
time 5-19
time function 5-172
__TIME__ preprocessor symbol 4-2
time.h file 5-1, 5-19
TMP_MAX 5-16
toascii function 5-173
_tolower function 5-175
tolower function 5-174
_toupper function 5-177
toupper function 5-176
Transcendental functions — see Mathematical functions 5-5

Type
casting 3-16
conversion 3-15
qualifiers 3-18
const 3-18
volatile 3-18
typedef keyword 2-5, 3-4, 3-19
Types
(see Data types)
enumeration 6-16
returned for functions 3-25, 8-3
user-defined 3-19

U

Unary expressions 3-8
#undef directive 4-5, 4-9, 4-24
ungetc function 5-178
union keyword 2-5
Union operators 3-13
Union return value 7-3
union type 3-2, 6-3
Unions 6-20
_UNIX preprocessor symbol 4-3
unlink function 5-179
unpacked
keyword 2-5
type 6-24
Unreachable (dead) code elimination 9-3
Unreachable code optimization 9-2
unsigned char type 3-2, 6-3, 6-4
unsigned int type 6-3, 6-7
unsigned keyword 2-5
unsigned long int type 6-3, 6-8
unsigned long type 3-2
unsigned short int type 6-3, 6-6
unsigned short type 3-2
unsigned type 3-2
/unsignedchar option (VMS) 6-4
Unused definition elimination optimization 9-4
-upd option (UNIX/ DOS) 8-5
User-defined types 3-19
User-modified routines
for embedded systems 8-14
for HP 64000 system 8-17
Usual arithmetic conversions 3-15

V

va_arg macro 5-180
va_end macro 5-182
va_start macro 5-183
Variable argument
functions
va_arg 5-12
va_end 5-12
va_start 5-12
types
va_list 5-12
Variables 3-1
allocation of variables 6-25
initializations 8-52
local 6-25
in the function prologue 7-6
names 7-10, 10-1
external 10-1
inside asm 8-4
public 10-1
static 10-2
use of back quote 8-4
saving initialized variables in ROM 8-52
sharing 3-6
static 5-21, 6-25
_VAX preprocessor symbol 4-3
vfprintf function 5-184
_VMS preprocessor symbol 4-3
void
keyword 2-5
type 3-3
volatile
keyword 2-5, 8-3
type 3-18
use with asm 8-3
vprintf function 5-185
vsprintf function 5-186

W

Warning messages C-1
while
    keyword 2-5, 3-19
    statement 3-24
write function 5-187

Z

zalloc function 5-21, 5-188
zerovars section 6-25

Reader's Response

Please complete and return the following Reader's Response form to help us improve our documentation. Your comments and suggestions are always appreciated.

Product Name _____ Manual Date/Version _____

Your Name _____ Title _____

Company _____ Address _____

Phone Number _____ _____

What is your level of programming experience? In this language?

_____Advanced _____Advanced

_____Intermediate _____Intermediate

_____Beginner _____Beginner

Circle the number which best expresses your rating of the information in this manual (*5 is the highest rating*):

Complete?	1	2	3	4	5	Easy to understand?	1	2	3	4	5
Accurate?	1	2	3	4	5	Helpful examples?	1	2	3	4	5
Well organized?	1	2	3	4	5	Helpful procedures?	1	2	3	4	5
Easy to Find?	1	2	3	4	5	At an appropriate level?	1	2	3	4	5
Easy to Read?	1	2	3	4	5	Overall rating?	1	2	3	4	5

3.) Which programming language do you use?

4.) Does the manual provide all the necessary information? If not, what is missing?

5.) Are more examples needed? If so, where?

6.) Did you have any difficulty understanding descriptions or wording? If so, where?

7.) Which sections of this manual are the most important? The most helpful?

8.) What are the best features of this manual?

9.) What are the areas in this manual that need improvement?

10.) Did you find any errors in this manual? (Specify section and page number.)

Software Performance Report Instructions

If problems or errors are encountered when using Microtec Research's software products (including documentation), a Software Performance Report (SPR) should be completed and returned to:

Microtec Research
Software Performance Reports
2350 Mission College Boulevard
Santa Clara, CA 95054

Any comments or suggestions you may have should also be included. All SPRs are directed to our Technical Support Department for analysis and response.

When completing the top section of the SPR, enter the following:

1. Product name and Version number as it appears on the media label or the banner that appears on the screen when the product is invoked.
2. Serial Number as it appears on the media label.
3. Name of the person (Customer) that Microtec Research can contact.
4. Name of the company that the software was Purchased From.
5. Company Name and a shipping Address.
6. Phone Number and/or FAX Number of contact name.
7. Host (machine where the Microtec Research product is run).
8. Host Operating System and Version Number.
9. Report Type showing the type of problem reported.
10. Customer Impact indicating the severity of the problem.

The following guidelines should be followed when completing the bottom section of the SPR:

1. Give as complete a description as possible of the problem, error, or suggestion, including the exact wording of any error messages.
2. If possible, isolate the problem by providing a small example.
3. If the error example is longer than one page of source code, send all information on a tape or floppy disk.
4. Carefully label any media you send and indicate whether you want it returned to you.
5. Include command files, listings, load maps, include files, data files, and all other relevant material with the SPR.
6. If applicable, send sample input and output listings.
7. If the output is from a compiler, send a mixed (source/assembly code) listing.

If the information on the SPR is complete and concise, it will help us give you accurate and timely service to any software problem.

Software Performance Report

Product:	Customer:
	Company Name/Address
Version:	
Serial Number:	
Purchased From:	Phone Number

Host: [ ] Apollo [ ] HP [ ] PC [ ] Other _____
[ ] Sun-3 [ ] Sun-4/SPARC [ ] VMS

Operating System _____ Version _____

Report Type:
- [ ] Problem/Error
- [ ] Documentation
- [ ] Feature Request
- [ ] Other _____

Customer Impact:
- [ ] Heavy
- [ ] Moderate
- [ ] Minor

Problem Description (Attach additional pages if needed):

Problem Recreation (Commands or lines of code used to duplicate problem):

Workaround (Attach additional pages if needed):

Microtec Research Inc.

2350 Mission College Blvd.

Santa Clara, CA 95054

Tel. 408.980.1300

Toll Free 800.950.5554

FAX 408.982.8266